TECHNISCHE UNIVERSITÄT WIEN

Institut für Computertechnik
Institute of Computer Technology

Master's Thesis

submitted by

Lukas Baischer

Registration Number 01426506

FPGA Based Embedded Neural Network Object Detector

In partial fulfillment of the requirements for the degree of

Diplom-Ingenieur (Dipl.-Ing.)

Vienna, Austria, 27.10.2021

Study code:	066 504
Field of study:	Embedded Systems
Supervisor:	Univ.Prof. Dipl.-Ing. Dr.techn. Axel Jantsch

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Copyright (C) 2021 Lukas Baischer

If you find this work useful, please cite it using the following BibTeX entry:

```
@Thesis{baischer2021,
 type        = {Master's Thesis},
 author      = {Lukas Baischer},
 title       = {FPGA Based Embedded Neural Network Object Detector},
 school      = {Vienna University of Technology (TU Wien)},
 year        = {2021},
 address     = {Gusshausstrasse 27--29 / 384, 1040 Wien},
 month       = {October},
}
```

Contact me:

lukas.baischer@student.tuwien.ac.at

This thesis is licensed under the following license: Attribution 4.0 International (CC BY 4.0)

You are free to:

1. Share — Copy and redistribute the material in any medium or format
2. Adapt — Remix, transform, and build upon the material for any purpose, even commercially.

This license is acceptable for Free Cultural Works.

The licensor cannot revoke these freedoms as long as you follow the license terms.

The entire license text is available at: https://creativecommons.org/licenses/by/4.0/legalcode

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Acknowledgements

I would first like to thank my thesis supervisor Univ.Prof. Dipl.-Ing. Dr.techn. Axel Jantsch, of the Institute of Computer Technology at Vienna University of Technology, for his patient guidance, enthusiastic encouragement and useful critiques of this research work. He consistently allowed this thesis to be my own work, but steered me in the right direction whenever they thought I needed it. I would also like to thank LARsys-Automation GmbH for funding and supporting my diploma thesis. Finally, I must express my very profound gratitude to my entire family and to my girlfriend for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Abstract

Deep learning enables machines to solve complex tasks that previously could only be mastered by humans. That led to breakthroughs in computer vision, speech recognition, and a wide variety of prediction models. A sub-area of computer vision is object detection. It determines in an image which objects are present from a pool of objects and where they are. There is an enormous number of possible applications for object detection in a wide variety of fields. Examples of that are autonomous driving, domestic help, service-, and assistance robots. In addition, there are industrial applications such as sorting tasks, quality checks, and other simple repetitive activities. However, the enormous computational effort that CPUs cannot handle in real-time is a crucial disadvantage of object detection based on deep learning compared to conventional algorithms. Therefore, real-time deep learning object detection requires additional hardware that accelerates the calculations. There are three possible platforms for accelerating, neural networks GPUs, FPGAs, and ASICs. The most common platform is the GPU. GPUs have the benefit that they are easy to use, enable training and inference, and achieve high throughput. One disadvantage of GPUs is that they have high energy consumption. Compared to GPUs, FPGAs and ASICs represent more energy-efficient alternatives. In addition, FPGAs and ASICs are optimizable for the respective task. Besides, FPGAs and ASICs enable various interface types and fieldbuses for connecting other machine parts. Compared to ASICs, FPGAs offer more flexibility because they consist of programmable hardware. The disadvantage, however, is that FPGAs achieve a lower throughput compared to ASICs and GPUs. Since FPGAs provide high flexibility in combination with low power consumption, FPGAs are a promising candidate for implementing a real-time capable object detector for edge devices.

Therefore, this work shows how to implement a flexible hardware accelerator in a low-cost FPGA that has similar usability as GPUs, low power consumption, and achieves a frame rate greater than 10 fps using YOLOv3-tiny. A converter makes it possible to quantize an existing CNN model from PyTorch and translate it into commands understandable by the hardware accelerator. In addition, the hardware accelerator is connected to a Linux system via a device driver that runs on the CPU part of the FPGA. That enables easy integration of the object detector to other program parts and external devices. In addition, the hardware accelerator offers the option of using different clock frequencies to find the most suitable ratio of energy consumption and the required frame rate for each application. If the hardware accelerator uses a clock frequency of 100MHz, it achieves a frame rate of approximately 9 fps. The entire Zybo-Z7-20 board consumes only 3.12 W. The frame rate can be increased up to 12 fps by increasing the clock frequency.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Kurzfassung

Deep Learning ermöglicht es Maschinen komplexe Aufgaben zu lösen, die davor nur von Menschen bewältigt werden konnten. Das hat vor allem in der Computer Vision, der Spracherkennung und in verschiedensten Voraussagemodellen zu Durchbrüchen geführt. Ein Teilbereich der Computer Vision ist die Objekterkennung, welche in einem Bild ermittelt, welche Objekte aus einem Objektepool vorhanden sind und wo sie sich befinden. Dafür gibt es eine enorme Anzahl an Anwendungsmöglichkeiten in den verschiedensten Bereichen. Beispiele dafür wären etwa autonomes Fahren, Haushaltshilfen, Service- und Assistenzroboter und vor allem auch Industrieanwendungen wie zum Beispiel Sortieraufgaben, Qualitätsüberprüfungen und jegliche andere einfache sich wiederholende Tätigkeiten die Sinnerfassendes Sehen benötigen. Allerdings hat die Objekt Erkennung basierend auf Deep Learning im Vergleich zu herkömmlichen Algorithmen einen Entscheidenden Nachteil, den enormen Rechenaufwand der von CPUs nicht in Echtzeit zu bewältigen ist. Aus diesem Grund benötigt eine echtzeitfähige Deep-Learning-Objekterkennung zusätzliche Hardware die die Berechnungen beschleunigt. Für diese Aufgabe gibt es drei mögliche Plattformen, GPUs, FPGAs und ASICs. Die am häufigsten verwendete Plattform stellen GPUs dar, da sie einfach zu verwenden sind, sowohl Training als auch Inference ermöglichen und einen hohen Throughput ermöglichen. Ein Nachteil von GPUs ist, dass sie einen hohen Energieverbrauch haben. Im Vergleich dazu stellen FPGAs und ASICs eine Energieeffizientere Alternative dar. Außerdem können FPGAs und ASICs für die jeweilige Aufgabe optimiert werden und es können zur Anbindung an andere Teile der Maschine alle Arten von Interfaces und Feldbusse hinzugefügt werden. Dabei bieten FPGAs die größere Flexibilität, da diese über programmierbare Hardware verfügen. Der Nachteil ist jedoch, dass FPGAs im Vergleich zu ASICs und GPUs einen geringeren Throughput erreichen. Wegen des geringen Energieverbrauchs und der hohe Flexibilität sind FPGAs ein vielversprechender Kandidat für die Implementierung einer echtzeitfähigen Objekt Erkennung in Edge-Devices.

Aus diesem Grund zeigt diese Arbeit wie in einem low-cost FPGA ein flexibler Hardwarebeschleuniger implementiert werden kann, der aufgrund eines Converters ähnlich einfach zu verwenden ist wie eine GPU, einen geringen Energieverbrauch aufweist und mit YOLOv3-tiny eine Framerate größer 10 fps erreicht. Der Converter ermöglicht es ein bestehendes CNN-Model in PyTorch zu quantisieren und voll automatisch in für den Hardwarebeschleuniger verständliche Befehle zu übersetzen, was eine einfache Verwendung garantiert. Außerdem ist der Hardwarebeschleuniger über einen Device Treiber an ein Linux angebunden, dass auf dem CPU-Teil des FPGAs ausgeführt wird. Dies ermöglicht eine einfache Anbindung der Objekterkennung an andere Programmteile und externe Geräte. Zusätzlich bietet der Hardwarebeschleuniger die Möglichkeit unterschiedliche Taktfrequenzen zu verwenden um das für die Anwendung am besten geeignete Verhältnis von Energieverbrauch und benötigter Framerate zu finden. Wenn eine Taktfrequenz von 100MHz verwendet wird erreicht der Hardwarebeschleuniger eine Framerate von ca. 9 fps. Dabei benötigen das gesamten Zybo-Z7-20 Board nur 3.12 W. Durch erhöhen der Taktfrequenz kann die Framerate auf bis zu 12 fps gesteigert werden.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Contents

Abstract **4**

Kurzfassung **5**

1 Introduction **15**

1.1 Contribution, novelties, and similarities of the hardware accelerator . . . 16

1.1.1 Contribution . . . 16

1.1.2 Novelties . . . 16

1.1.3 Similarities to related works . . . 17

1.2 Scope of the thesis . . . 17

1.3 Performance evaluation metrics . . . 17

2 Neural Networks **19**

2.1 Fully connected network . . . 20

2.2 RNN . . . 20

2.3 CNN . . . 21

2.4 DNN compression methods . . . 22

2.4.1 Pruning . . . 22

2.4.2 Tensor decomposition . . . 23

2.4.3 Compact models . . . 24

2.4.4 Knowledge distillation . . . 25

2.4.5 Quantization . . . 25

2.5 Algorithmic optimization . . . 27

2.5.1 General matrix multiplication . . . 27

2.5.2 Winograd transformation . . . 28

2.5.3 Fast fourier transformation . . . 30

2.6 Accelerating nested loops . . . 30

2.6.1 Loop reordering . . . 31

2.6.2 Loop unrolling . . . 31

2.6.3 Loop pipelining . . . 31

2.6.4 Loop tiling . . . 31

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

3 CNN Based Object Detection **32**
3.1 Mean average precision . . . 33
3.2 R-CNN . . . 33
3.3 SSD . . . 34
3.4 YOLO . . . 34
3.5 RetinaNet . . . 34
3.6 Comparision of available object detecors . . . 34

4 Neural Network Hardware Accelerator **36**
4.1 Graphics processing unit . . . 37
4.1.1 Frameworks . . . 38
4.1.2 Architecture . . . 38
4.1.3 Embedded GPUs . . . 38
4.1.4 Overview . . . 38
4.2 Application specific integrated circuits . . . 39
4.2.1 DianNao family . . . 40
4.2.2 EIE: Efficient inference engine . . . 40
4.2.3 Eyeriss . . . 40
4.2.4 Tensor processing unit . . . 40
4.3 Field programmable gate arrays . . . 41
4.3.1 Algorithmic optimization in FPGA based hardware accelerators . . . 41
4.3.2 Mapping convolutional neural networks on FPGAs . . . 41
4.3.3 Hardware architectures . . . 42
4.3.4 Overview of existing implementations . . . 42
4.4 Comparison . . . 45
4.4.1 Throughput . . . 46
4.4.2 Latency . . . 46
4.4.3 Accuracy . . . 46
4.4.4 Power . . . 46
4.4.5 Usability . . . 47
4.4.6 Fields of application . . . 47

5 Neural Network Hardware Accelerator Design **49**
5.1 Data streamer . . . 50
5.1.1 Tile recurrence . . . 51
5.1.2 Padding . . . 52
5.2 Dataflow control . . . 52
5.2.1 Available and possible future operation modes . . . 55
5.3 Processing element . . . 57
5.4 Processing unit int8 . . . 58
5.5 Processing unit float8 . . . 59

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

5.6 Data collector . . . 60
5.7 Absolute maximum throughput . . . 62
5.8 Maximum achievable PE utilization . . . 62

6 Software Toolchain **64**
6.1 Quantization aware training . . . 65
6.2 Model converter . . . 66
6.2.1 Feature map tiling . . . 67
6.3 Embedded software . . . 67
6.3.1 Hardware accelerator driver . . . 68

7 Performance Evaluation **70**
7.1 Quantizing YOLOv3-tiny . . . 70
7.2 Measuring throughput and latency . . . 72
7.3 Reasons for low PE utilization . . . 74
7.3.1 Possible improvements of design . . . 75
7.4 Energy efficiency . . . 75
7.5 FPGA resource utilization . . . 77
7.6 Comparison to related implementations . . . 78
7.7 Comparision to Nvidia GeForce MX250 GPU . . . 79

8 Conclusion **80**

9 Outlook **82**

Appendix A **90**
.1 Dataflow visualizations . . . 90
.2 YOLOv3-tiny application . . . 93

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

List of Figures

2.1 A single artificial neuron and the similarities to the human brain. 19
2.2 Simple neural network example and terminology. 21
2.3 Recurrent neural network. 21
2.4 Convolutional neural network principle. 22
2.5 Decomposition of weight tensors in CNNs. 24
2.6 Compact model design elements. 25
2.7 Batching to convert vector-matrix multiplication into matrix-matrix multiplication. 28
2.8 Winograd transformation. 28
2.9 Fast Fourier Transformation to accelerate CNNs. 31

3.1 Object detector principle. 32

4.1 Gap between required number of operations and performance density 37
4.2 Temporal single instruction multiple data (SIMD)/single instruction multiple threads (SIMT) architecture 38
4.3 Typical hardware architectures used in an FPGA. 43

5.1 Neural network hardware accelerator overview. 50
5.2 Block diagram data streamer. 51
5.3 Block diagram dataflow control. 53
5.4 Comparison of data flow using int8 and float8 implementation. 54
5.5 Data flow for 3x3 convolutions width stirde 2 using the float8 implementation. 55
5.6 Block diagram processing element int8 and float8 57
5.7 Processing unit int8. 58
5.8 Processing unit float8 implementation with floating point extension. 59
5.9 Block diagram data collector. 61

6.1 Software toolchain. 64
6.2 Model compiler workflow. 66
6.3 Overview of the software abstraction layers. 68

7.1 Example detection using the hardware accelerator and an example image with a size of 383×384 72
7.2 Achievable throughput of YOLOv3-tiny dependable of the PE-Clk frequency using a Mem-Clk frequency of 100 MHz and 125 MHz 73

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

7.3 Reason for low memory bus utilization. 74
7.4 Setup for measuring the power consumption of the Zybo-Z7-20 board 75
7.5 Power and energy efficiency using different frequencies for Mem-Clk and PE-clk 76
7.6 FPGA resource utilization of xc7z020clg400-1 using the int8 implementation 77
7.7 FPGA resource utilization of xc7z020clg400-1 using the float8 implementation 77
7.8 Comparison with related implementations in performance space 78
7.9 Performance and energy efficiency comparison of the presented hardware accelerator and an Nvidia GeForce MX250 GPU. 79

1 Dataflow for 1x1 convolutions 90
2 Dataflow for 5x5 convolutions 91
3 Dataflow for 5x5 convolutions width stirde 2 92

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

List of Tables

3.1 Comparison of most common object detectors. 35

4.1 Overview of GPU-based hardware accelerator implementations. 39
4.2 Overview of existing ASIC-based hardware accelerator implementations. 40
4.3 Overview of existing FPGA-based hardware accelerator implementations. 45

5.1 Comparison required transfer rate and number of operations. 63

6.1 Weight command list. 66
6.2 Feature-map command list. 67
6.3 Software platform 67

7.1 Tasks for quantizing YOLOv3-tiny 71
7.2 Latency, throughput, power consumption, energy efficiency and PE utilization 72

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Symbols

FLOP/s	floating point operations per second
fps	frames per second
GOPS	giga operations per second
mAP	mean average precision

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Acronyms

AI	artificial intelligence
ALU	arithmetic and logic unit
ANN	artificial neural network
ASIC	application specific integrated circuit
BNN	binarized neural network
BRAM	block-RAM
CLB	configurable logic block
CNN	convolutional neural network
CPU	central processing unit
DMA	direct memory access
DNN	deep neural network
DSP	digital signal processor
FC	fully connected
FF	flip flop
FFT	fast Fourier transformation
FIR	finite impulse response
FPGA	field programmable gate array
GEMM	general matrix multiplication
GPU	graphics processing unit
GRU	gated recurrent unit
HLS	high level synthesis
IC	integrated circuit
IP	intellectual property
LE	logic element
LSTM	long short term memory
LUT	lookup table
MAC	multiply and accumulate
MLP	multilayer perceptron
OpenCL	open computer language
PE	processing element
PU	processing unit

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

ReLU	rectified linear unit
RNN	recurrent neural network
RTL	register transfer level
SIMD	single instruction multiple data
SNN	spiking neural network
SSD	single shot multibox detector
TNN	ternary neural network
v4l2	video for linux 2
YOLO	you only look ones

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Chapter 1

Introduction

Deep learning is currently one of the most promising approaches to solve complex tasks in many different fields of application, which could only be solved by humans beforehand [1]. That is possible since Deep neural networks (DNNs) can consider an enormous number of learned parameters. Especially in applications such as computer vision or speech recognition, neural networks led to a breakthrough since they achieve higher accuracy and robustness compared to conventional algorithms and perform in some cases even better than human experts [1–3].
Neural networks use statistical learning over plenty of training data to extract high-level features from the input data. That leads to an efficient representation of the input space and a good generalization. In consequence, DNNs achieve higher accuracy in object classification and detection compared to conventional algorithms [2]. With the progress of deep neural networks used for computer vision a reliable, robust object detection is possible [4]. That enabled many other fields of application such as autonomous driving, automated optical inspection, and many other industrial applications [3, 5]. Outsourcing such applications to server farms is not reasonable because of real-time requirements and security reasons [3]. For this reason, DNN-based object detectors are increasingly implemented in embedded edge devices.

In recent years, the progress of increasing the accuracy of neural networks and object detectors bases on creating deeper and more complex neural networks, which leads to an exponentially growing network size [3]. Consequently, that leads to an exponentially increasing computational effort and memory size [3]. Therefore, central processing units (CPUs) can hardly handle the computations for using DNNs on embedded edge devices. For this reason, hardware accelerators parallelize the calculations for DNNs to increase the inference processing performance [2, 3]. There are three major hardware platforms for neural network hardware accelerators: Graphics processing units (GPUs), application specific integrated circuits (ASICs), and field programmable gate arrays (FPGAs).
Chapter 2 introduces the field of neural networks and shows the available algorithmic optimization and compression methods. Chapter 3 shows an overview of existing object detectors. Chapter 4 introduces neural network hardware accelerators, emphasizes why hardware accelerators are needed, and provides a detailed comparison of the three major hardware platforms. In a nutshell, GPUs offer high throughput, high accuracy, and excellent usability but have a high power consumption and are not well suited for real-time computing. ASICs are fully customizable, which enables them to achieve the best performance and energy efficiency. However, complete design freedom is associated with high

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

acquisition costs, a long time to market, and a complex design process. Compared to that, FPGAs provide high design freedom in combination with high energy efficiency, low initial costs, a shorter time to market, and less complexity. In addition, FPGAs provide good scalability due to many interchangeable devices. However, in terms of pure throughput GPUs, and ASICs outperform FPGAs. But FPGAs are highly suitable for embedded real-time devices, where latency and power efficiency are the most significant parameter.

In industry, there is a broad field of applications for embedded DNN-based object detection. That ranges from automated optical inspection, through sorting or counting of parts, to gripping objects with a robot arm. In contrast to conventional computer vision, industrial applications allow additional assumptions regarding illumination and possible scenarios. That can significantly increase the accuracy of the object detector since the training process covers all allowed scenarios. Therefore, the test accuracy comes closer to the training accuracy. However, industrial applications often have additional requirements, such as dependability, maintainability, fieldbus requirements, real-time requirements, energy, and space consumption.
Considering these requirements, FPGAs are suitable for building embedded object detection hardware for industrial applications, since FPGAs provide good energy efficiency, low space consumption, and provide a long-term availability of spare parts. FPGAs are reprogrammable hardware devices, which enable easy maintenance using online updates. Furthermore, the logic structure enables support for hard real-time. In addition, it offers the possibility to implement custom fieldbus, actuator and sensor interfaces, and fast digital controllers directly in hardware. For this reason, FPGAs are well suited for the implementation of an autonomous subsystem, which carries out the object detection and the associated control of manipulators in a single device.

1.1 Contribution, novelties, and similarities of the hardware accelerator

1.1.1 Contribution

There are currently many different approaches for implementing embedded object detectors. Considering only the throughput, GPUs and ASICs are the most feasible choice. However, in real-time object detection applications, which have a limited power or space budget, FPGA-based applications are a viable choice since FPGAs are optimizable for batch size 1, and provide a higher energy efficiency compared to GPUs. In addition, FPGAs provide the ability to implement an entire system in a single device, including fast controller, custom fieldbus, custom sensor interfaces, and custom actuator interfaces. The long-term availability of spare parts is particularly important for long-life industrial systems, which are considerably longer for FPGAs than for GPUs. However, most real-time capable FPGA-based object detector implementations published in related works require high-end FPGA devices. High-end FPGA devices are expensive, and therefore not competitive to GPU-based implementations. For this reason, the goal of this thesis is to implement a real-time capable object detector in a low-cost FPGA device, which provides competitive usability compared to GPUs.

1.1.2 Novelties

- **Processing unit structure:** Using a 48bit DSP slice in combination with a floating point extension, implemented in fabric, for computing three 8-bit floating point MAC operations in parallel or two 8-bit integer MAC operations in parallel.
- **Processing element structure:** Comparatively high data locality by using local block-memories in combination

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

with feature map tiling. For loops are not parallelized in the lowest level but rather in the Y-axis and the output channels, which simplifies the reuse of local data and the implementation of various operation modes.

- **Software toolchain:** Implementation of the complete design flow. Starting with quantization-aware training through automated tiling and command generation up to the connection to a Linux system through a kernel driver and a python wrapper.
- **Comparison of float8 and int8:** Comparison of a float8 and an int8 processing unit.
- **Asynchronous Mem-Clk and PE-Clk:** The hardware accelerator allows adjusting the Mem-Clk and the processing element (PE)-Clk asynchronously, which allows overclocking the PEs and thus a higher computing speed.

1.1.3 Similarities to related works

- **Processing unit structure:** The int8 implementation is similar to the int8 deep learning on Xilinx DSP Slices approach. However, the proposed float8 processing unit (PU) has a speed advantage of $1.5\times$ because of the 3-fold parallelism compared to the 2-fold parallelism preposed by Xilinx [6].
- **Floating point extension:** Comparable to brainwave. However, brainwave uses shared exponents for 128 numbers and a completely different hardware architecture [7].
- **Processing element arrangement and dataflow:** Angel-Eye uses a similar top-level hardware accelerator architecture [8]. However, due to the different data flow, the local PE caches, the higher clock frequency and the different parallelization strategy a superior throughput can be achieved, even if no parallelization of the digital signal processors (DSPs) would be used.
- **Object detector in low cost FPGA:** Various implementations of object detectors in high end FPGAs are already existing. However, this work compares two quantization methods, introduces a new dataflow, and shows an entire design flow from low-level processing unit design to a high level python wrapper.

1.2 Scope of the thesis

The goal of this thesis is to evaluate if it is possible to implement an embedded object detector that can perform the inference of YOLOv3-tiny with a frame rate of 10 in the low-cost commercial FPGA-board *Zybo-Z7-20*. The performance of the implemented hardware accelerator is compared with related FPGA implementations and with a low-cost laptop GPU to determine which hardware is better suited for the given real-time object detection task. In addition, the hardware accelerator should support various neural network and object detector structures, which enable comparable usability to GPUs. In addition to the hardware accelerator, a camera interface is to be implemented. Optionally an HDMI transmitter interface to visualize the results is to be implemented if enough resources are available.

Research question:
Is it possible to implement an object detector based on YOLOv3-tiny with a frame rate of 10fps in a low-cost Zybo-Z7-20 FPGA-board, and how does it perform compared to a low-cost laptop GPU?

1.3 Performance evaluation metrics

The performance of the system is determined using the following commonly used parameters:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

- ***Accuracy***. Indicates the quality of results for a given task [2]. It is expressed in the percent of correct detected elements. The accuracy depends on the used dataset. For this reason, a fair comparison of the accuracy is only possible if the same dataset is used [2]. For the accuracy, two metrics can be used: Training accuracy and testing accuracy. Throughout this thesis, "accuracy" is used for the testing accuracy since it indicates the generalisability of the network and therefore how it will perform on real input data [9].
- ***Throughput***. Classifications or detections per second [9]. Analysis of high volume data [2]. The computational effort of different models is considered by multiplying the throughput with the required number of operations for a single classification or detection.
- ***Latency***. End-to-end processing time for one classification or detection pass [9].
- ***Energy efficiency***. Throughput per unit power [9].
- ***Hardware cost***. On chip storage, number of cores, chip area + manufacturing process technology [2]. In terms of FPGAs: Utilization of lookup tables (LUTs), flip flops (FFs), block-RAM (BRAM),DSPs etc..

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Chapter 2

Neural Networks

Neural networks are a part of the broad field of artificial intelligence (AI), which aims to solve tasks similar to humans. In general, AI deals with building intelligent machines [2, 10]. Neural networks are a sub-area of AI which focuses on finding algorithms that mimic the functionalities of the human brain.

Figure 2.1 visualizes the basic element of neural networks, a single neuron. It shows the connection to the human neuron and the resulting mathematical representation, which is also shown in Equation (2.1). In Equation (2.1) x_i, w_i, $f(\cdot)$, b and y_j are the input activations, weights, non-linear activation function, bias and output activation [2, 10]. Equation (2.1) is the basis for all neural network types described hereafter.

$$y_j = f\left(\sum_i w_i x_i + b\right) \tag{2.1}$$

For the non-linear activation function $f()$ any non-linear function is conceivable. Early neural networks mainly use the

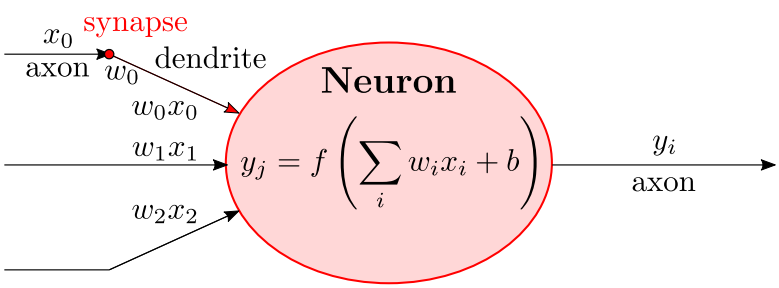


Figure 2.1: A single artificial neuron and the similarities to the human brain. x_i, w_i, $f(\cdot)$, b and y_j are the input activations, weights, non-linear activation function, biases and output activations. Adopted from [10, Fig. 1].

Sigmoid function. Later the DNNs mostly use rectified linear unit (ReLU) [2]. ReLU cuts off all negative values, which has the advantage of low computation effort and achieves better accuracy on convolutional neural networks (CNNs) in comparison to Sigmoid [11]. However, recent CNN for object detection use LeakyReLU and Mish, since they preserve the information included in negative activations [4, 12].

In a neuronal network, single neurons are arranged in layers. The arrangement of the neurons inside a layer and the connection of neurons between the layers define the layer type. The used layer types define the type of the neural

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

network. The three most common neural network types are:

- Fully connected (FC) or feed-forward neural networks
- Recurrent neural networks (RNNs)
- Convolutional neural networks (CNNs)

Historically fully connected neural networks are the first form and provide the basic learning functionality. However, modern neural networks use fully connected layers as part of CNNs or RNNs. In some literature FC-networks are called multilayer perceptron (MLP) or Dense networks. The name Dense-network derives from the high parameter density of fully connected neural networks. CNNs consider the position of a neuron in a layer in addition to the activation value. That ability suits CNNs for use in computer vision applications. Compared to FC-networks, CNNs require less storage since all weights in a filter stay constant for a single layer. Tasks, which require information of previous events such as speech recognition or in the financial world use RNNs. RNNs consider the information of former runs by using feedback loops between layers. DNN does not describe a specific neural network structure. The keyword is related to deep learning, which indicates that it is a particularly deep network, i.e. a network with many layers [2, 10].

2.1 Fully connected network

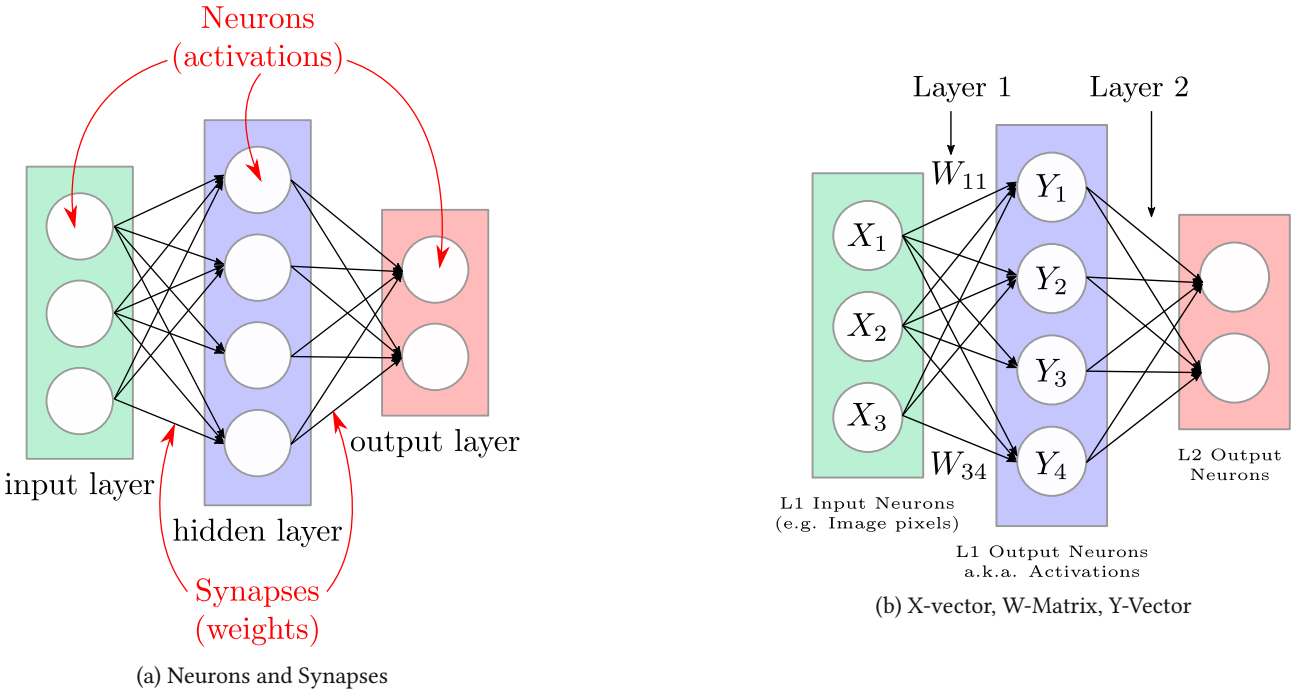
Figure 2.2 visualizes the basic layer types and the terminology of fully connected networks. The first layer in a network, which receives the data that contains the information to be analyzed, is called the input layer. The inputs of a network can be, for example, an image or a sampled audio sequence. The layers followed by the input layer process the input data. These layers are called hidden layers. The last layer of a neural network is the output layer. It provides the results of a neural network. In fully connected networks, each output neuron represents the probability of a specific outcome. Each layer of a fully connected network is described by Equation (2.1), whereas the weight matrix w_i contains an entry for each neuron connection between two layers [10].

2.2 RNN

In contrast to FC-neural networks and CNNs, recurrent neural networks (RNNs) use intralayer recurrent connection. These recurrent connections enable RNNs to consider past results or states in addition to the current input data [2, 13]. Due to this ability, RNNs are particularly suitable for tasks like speech recognition, translation, and financial predictions [10].

Figure 2.3 visualizes a simple recurrent neural network with a recurrent connection at the output layer. The output of the network is a trace back to the input of the output layer. That enables RNNs to consider the previous output of the neural network. Popular examples for recurrent layer are long short term memory (LSTM) or gated recurrent unit (GRU) layer [13].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN



(a) Neurons and Synapses

(b) X-vector, W-Matrix, Y-Vector

Figure 2.2: Simple neural network example and terminology. (a) shows a simple example of a fully connected neural network. (b) gives an overview how the X-Vector, W-matrix and Y-vector are interpreted. Adopted from [10, Fig. 2].

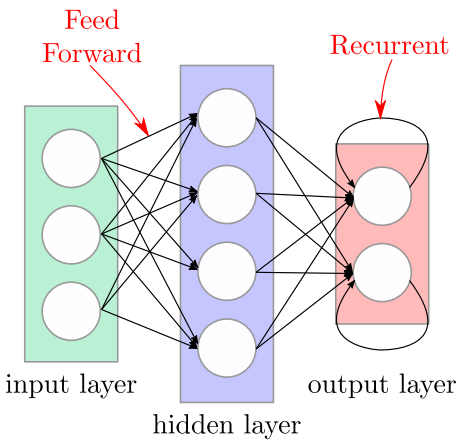


Figure 2.3: Simplified structure of a recurrent neural network [2, Fig. 8].

2.3 CNN

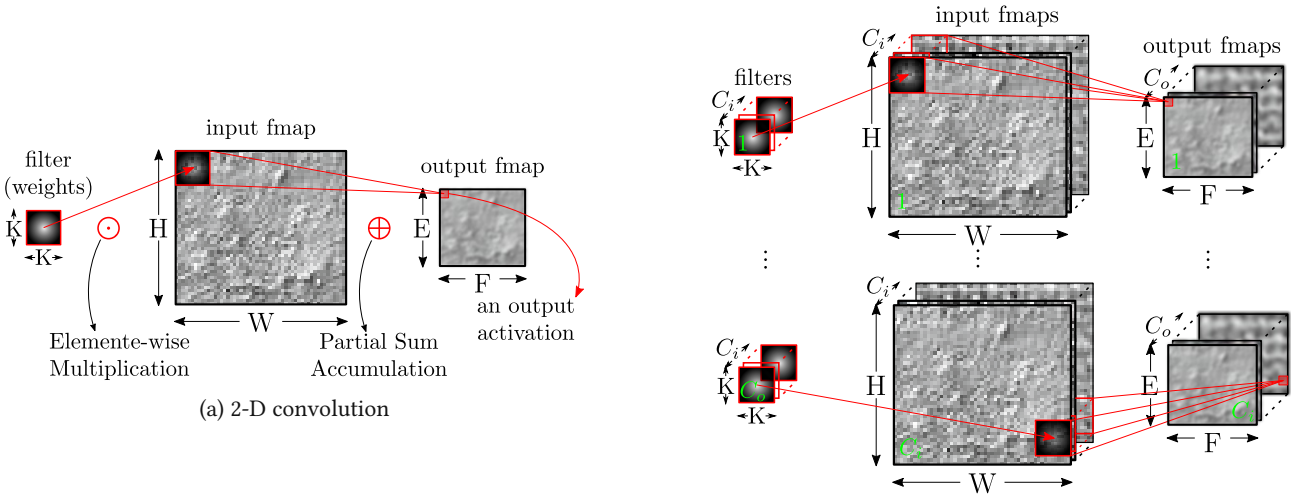
In contrast to FC-networks and RNNs, Convolutional neural networks (CNNs) consider the geometric information included in the position of each pixel by using convolution operations. That ability makes CNNs state of the art for images classification and detection tasks [2, 13–16]. The elementary function of convolutional layer is the 2-D convolution, which extracts features for an images. That enables CNNs to do a reliable classification of images [11]. A 2-D convolution moves a filter with size $k \times k$ along an image or feature map producing an output pixel for each output channel at each step. Figure 2.4a visualizes the behavior of a 2-D convolution. It shows that a 2-D convolution performs an element-wise multiplication of the filter with the overlapping image tile at each step. After that, it accumulates the results of the element-wise multiplication. Equation (2.2) shows the mathematical representation of a 2-D convolution [2, 17, 18]. $conv2D$ computes a convolution for a 2-D input tensor $X^{conv}[C_i]$ and a 2-D weight kernel $\Theta[C_o, C_i]$ by summing the results of the element-wise multiplication $\odot$ of the overlapping elements of the 2-D input tensor and the weight kernel.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

$$conv2D\left(X^{conv}[C_i], \Theta[C_o, C_i]\right) = \sum X^{conv}[C_i] \odot \Theta[C_o, C_i] \tag{2.2}$$

CNNs apply multiple filters in parallel to create complex high-level feature maps. Figure 2.4b visualizes that the use of C_o filters on a single input image creates C_o output channels. All output channels together form a feature map. For the computation of a single output channel, the 2-D convolution accumulates the results of all C_i input channels. After that, it applies a non-linear function $f(\cdot)$. The weights of each filter are different for each channel. That leads to $C_i \times C_o$ filters for a single convolutional layer [10]. Equation (2.3) visualizes the computation of an output feature map of a convolutional layer [17]:

$$Y^{conv}[C_o] = f\left(\sum_{c_i=0}^{C_i} conv2D\left(X^{conv}[c_i], \Theta[C_o, c_i]\right)\right) \tag{2.3}$$



(a) 2-D convolution

(b) Multidimensional convolution

Figure 2.4: Convolutional neural network principle. (a) shows the principle of a 2-D convolution used in image processing. (b) visualizes the principle of convolutions in high dimensional CNNs. Adopted from [10, Fig. 3].

2.4 DNN compression methods

A convolutional layer requires $C_o \cdot C_i \cdot W \cdot H \cdot K \cdot K$ multiply and accumulate (MAC) operations which leads to significant computational effort for DNNs. There are several approaches to reduce the required computation effort for deep CNNs. On the one hand, pruning and compact models are used to create more efficient neural network structures, on the other hand the bit width of weights and activations is minimized in order to use more efficient hardware [2, 3, 9, 10].

2.4.1 Pruning

Pruning searches for those weights that have little or no influence on the result. Eliminating these weights leads to little or no accuracy loss. That reduces the number of required computations, the disk storage, and the overfitting of the network [3, 19–21]. In some literature, pruned networks are called sparse networks [20].

The main difficulty of pruning is to determine the importance of weights or filters and thus to decide whether pruning is permitted or not. The absolute value of weights proves to be a proper measure [22, 23]. Pruning can lead to an accuracy loss. Therefore, pruning requires some retraining steps [22, 23]. However, when pruning a large portion of

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

the connections, an issue with retraining can be that the gradient vanishes, which leads to slow convergence. Pruning convolutional and fully connected layers separately reduces the side effect of vanishing gradients. Another method to avoid vanishing gradients is to slow down pruning [22, 23].
Pruning in various granularity is possible: fine-grained pruning, vector-level pruning, kernel-level pruning, group-level pruning, and filter-level pruning [21].

Fine-grained pruning

Fine-grained pruning has no constraints, it permits pruning any weight which is found to be unimportant [21]. Han et al. introduce Deep Compression, a Fine-grained pruning method. It starts with a pre-trained network. Next, Deep Compression prunes all weights below a certain threshold. The last step is to retrain the remaining network. Deep Compression reduces the parameter number of AlexNet by 9× and the parameter number of VGG-16 by 13× without any drop in accuracy [20]. However, pruning requires caution, since pruned weights cannot be reestablished and incorrect pruning leads to severe accuracy loss [22]. Therefore, Guo et al. introduce splicing. It reconnects pruned connections, that show to be important. Splicing is similar to the human brain since pruning and splicing are analogical to the synthesis of excitatory and inhibitory neurotransmitters in the human nervous system [22]. Using pruning and splicing the dynamic network surgery algorithm iteratively compresses the network. It leads to a 17.7× compression rate using AlexNet, with a minor accuracy loss of 0.33% [22].

Vector-level and kernel-level pruning

Vector-level and kernel-level pruning constraints pruning to vectors or filters of convolutional layer [21]. Compared to fine-grained pruning, vector-level and kernel-level pruning are more hardware friendly since hardware accelerators are optimized for vector operations and therefore only benefit slightly from fine-grained pruning [10, 21].

Group-level pruning

Group-level pruning merges filters of a convolutional layer which operate on the same input feature map and provide the same pattern [21]. Based on that Lebedev et al. introduce the group-wise brain damage algorithm for group-level pruning. It achieves a 3.2× speed up for convolutional layers of AlexNet [24].

Filter-level pruning

Filter-level pruning eliminates channels of a convolutional layer, that are found to have a low influence on the results. The feature map of the following layer indicates which channel can be pruned [21].

2.4.2 Tensor decomposition

As mentioned before, the fundamental operations for training and inference of neural networks are tensor (including matrix) computations. Therefore, tensor decomposition methods enable compression and acceleration of neural network models [13]. Tensor decomposition has two advantages. First, it decomposes a large tensor into two smaller tensors that fit better into local memory. Second, tensor decomposition reduces the computational effort by eliminating redundancies when the original tensor is not of full rank [10].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Tensor Decomposition for CNNs

Tensor decomposition assumes that the output produced by $\mathbf{y} = W\mathbf{x}$ can be approximately replicated by a low-rank subspace [25]. Figure 2.5 visualizes the low-rank decomposition. It creates a low-rank subspace by replacing $\mathbf{y} = W\mathbf{x}$ with $y = PW'\mathbf{x} + b$. That reduces the computational complexity from $O(dk^2c)$ to $O(d'k^2c) + O(dd')$, with c is the input channel size, d is the original output channel size and d' is the intermediate channel size [25]. Zhang et al. show that tensor decomposition reduces the required number of computations by a factor of 2.9 with a minor accuracy loss of 0.3% [25].

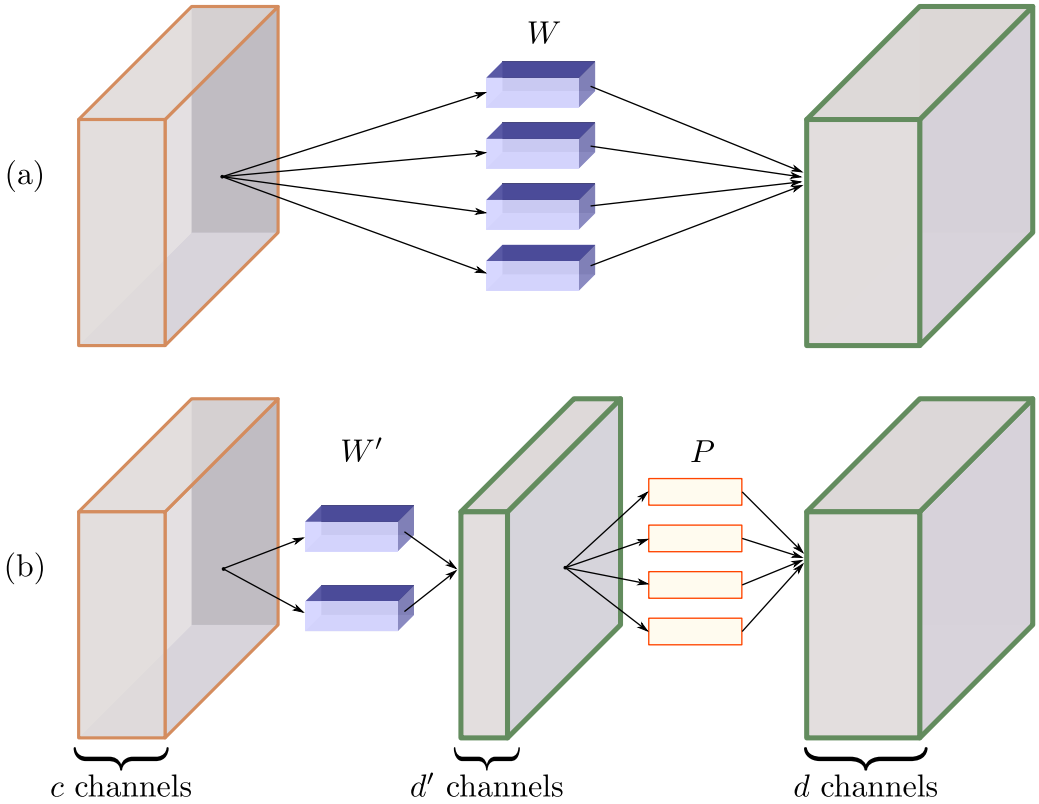


Figure 2.5: Decomposition of weight tensors in CNNs. (a) original layer with complexity $O(dk^2c)$. (b) approximated layer with compexity reduced to $O(d'k^2c) + O(dd')$. Figure adopted from [10, Fig. 6].

2.4.3 Compact models

Compact models use the opposite approach. Instead of creating sparse models using pruning and tensor decomposition, compact models use sparse neural networks as the starting point for training. Compact models aim to replace large convolutional kernels with smaller ones. For example, replacing a 5×5 convolution with two 3×3 convolutions requires 28% fewer parameter [13]. However, the computational effort of a 3×3 convolution is still high. Therefore, Chollet et al. introduce depthwise separable convolutions, which consist of 1×1 convolutions followed by depthwise separated 3×3 convolutions [26]. The 1×1 convolutions observe the correlation of the feature map channels, while the 3×3 convolutions act as a conventional filter in the x- and y-direction. That method is similar to tensor decomposition[1], except that the depthwise separable convolutions define that structure before training.

He et al. introduce residual layer, which reuses the information of features generated by a former layer by accumulating it to the current feature map [14]. That enables the creation of particularly deep networks, which yields an accuracy

[1] See Section 2.4.2

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

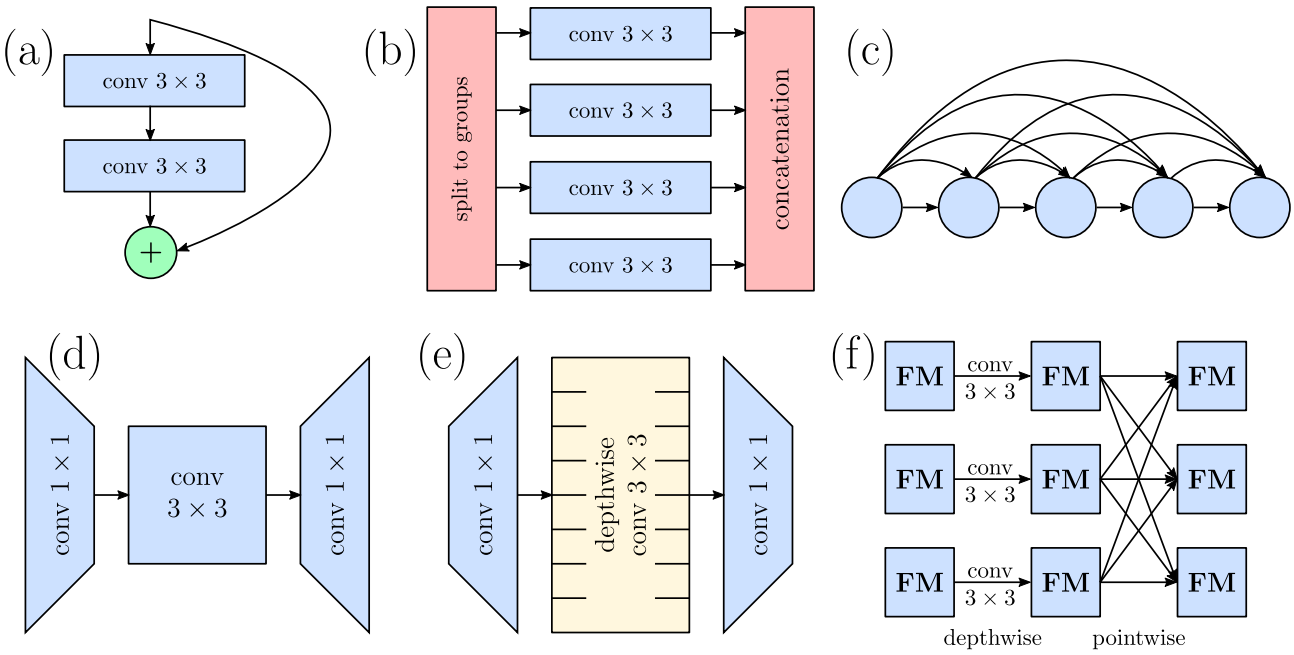


Figure 2.6: Compact model design elements. (a) Residual connection. (b) Group convolution. (c) Densely connected block. (d) Bottleneck layer. (e) Inverse bottleneck layer. (f) Depthwise separable convolution. Adopted from [10, Fig. 7].

gain [13].

Figure 2.6 visualizes how compact models realize these approaches. Figure 2.6a shows a residual connection. Figure 2.6b shows a group convolution, which splits the channels into group-wise separated channels. Figure 2.6c shows a densely connected block. Figure 2.6d shows a bottleneck layer, which downscales the channel number by using the less computational intensive 1×1 convolution layer. After the downscale, the computational costly 3×3 convolution layer uses less input and output channel. After that, the bottleneck layer expands the channel number using a 1×1 convolution layer. The inverse bottleneck layer visualized in Figure 2.6e does the opposite, but it uses depthwise separated 3×3 convolutions, which require a low filter number. Figure 2.6f visualizes the difference between depthwise separated convolutions and pointwise convolutions. It shows that depthwise convolutions consider only a single input channel for computing a new output channel. In most modern compact models these approaches are combined [12, 15, 16, 27].

2.4.4 Knowledge distillation

Different models approach a problem from different directions. Therefore, each model has different strengths and weaknesses. For this reason, averaging the predictions of distinct models gives higher accuracy than using a single model. However, running multiple models to get a single result is computationally intensive. Knowledge distillation targets that problem by combining the knowledge of different networks into a single network. The original models are called the teacher, and the new model is called the student. Knowledge distillation adjusts the weights and biases of the student network based on the output of the teacher networks using stochastic gradient descent. Using knowledge distillation enables a simple student model to achieve an accuracy that would be unachievable using direct training [2].

2.4.5 Quantization

The bit width is limited in all digital devices, whereby the accuracy of the representation of a digital number is limited. Representing a number with a limited bit number is called quantization. In principle, the higher the bit width and thus

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

the higher the computing accuracy, the greater the hardware costs [2]. Due to the arbitrary exponent, floating-point quantization achieves higher precision compared to fixed-point quantization. However, floating-point operations require more complex arithmetic compared to fixed-point operations. That leads to higher power consumption, resource costs, or latency [9]. Therefore neural network hardware accelerators heavily use fixed-point quantization to reduce power consumption and resource costs [8, 9, 19, 28–36].

Quantizing a neural network means using a quantization method with reduced resource cost and a less accurate representation of numbers. However, the goal is to minimize the quantization error while lower hardware costs [2]. Since the computations of individual layers, channels, and pixels are encapsulated operations, it is possible to use different types of quantization for layers, channels, and filters [2]. A quantization of both the weights and the activations is possible. Changing the bit width of weights has a smaller effect on the accuracy than changing the bit width of activations [2]. Neural networks allow both linear and non-linear quantization. The benefit of linear quantization is that it uses existing calculation methods, while non-linear quantization requires specialized computation methods. The advantage of non-linear quantization, however, is that it enables resolving frequently occurring number ranges with greater precision than less frequently occurring ranges. It is possible to determine the numerical ranges by observing the distribution of weights and activations in a neural network or by learning them e.g. using k-means clustering [2]. Processing units for non-linear computations usually use LUTs, since they offer the possibility to output a specific bit pattern depending on the input pattern.

Binarized quantization

Binarized quantization is an extreme version of fixed-point quantization. The idea is to constrain both weights and activations to $\{1, -1\}$ [37]. That is beneficial since hardware-friendly XNOR operations can replace costly multiplications [10, 38]. However, binarized quantization of both weights and activations leads to a heavy loss in accuracy [2, 38]. Therefore, retraining binarized neural network (BNN) is necessary to regain some accuracy. Another method to preserve accuracy is to use full precision for the first and/or the last layer [2, 10, 31].

Stochastic or deterministic methods are possible for quantizing BNNs. The most common function for quantizing BNNs is the $Sign(x)$ function [37, 38]:

$$x_b = Sign(x) = \begin{cases} +1 & x \geq 0 \\ -1 & x < 0 \end{cases} \tag{2.4}$$

Training a neural network requires the compuation of the gradient. However, the derivative $Sign(x)$ is a Dirac $\delta(x)$ function, which is not suitable for calculating the direction of the gradient in order to update the weight. Therefore, $tanh(x)$ or $Htanh(x)$ functions are used instead of the $Sign(x)$ [38].

Ternary quantization

In difference to binarized quantization, ternary quantization adds a bit to enable a representation of a zero weight [39]. Therefore, ternary neural networks (TNNs) can prune connections in the network. Which can increase the accuracy and decrease the inference latency significantly since zero weights can be skipped [39]. In the first version TNNs use

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

$\{-1, 0, +1\}$. However, some related works use constant values different from 1 to preserve more accuracy. An example is to use $\{-E, 0, +E\}$, with E is the mean absolute value of the learned weights. Another possibility is to use different constant values for each layer $\{-W_l^n, 0, +W_l^p\}$ based on the full-precision weights [40].

Ternary quantization reduces the model size by a factor of 16 in comparison to full-precision models [40]. Additionally $\{-1, 0, +1\}$ quantization requires no costly multiplication operations [39, 40]. In contrast to that, $\{-E, 0, +E\}$ and $\{-W_l^n, 0, +W_l^p\}$ quantization still requires multiplications. Nevertheless, it enables the use of constant multipliers. In both cases, it enables a more efficient hardware accelerator implementation in terms of speed, power efficiency, and area consumption, while tolerating a moderate accuracy loss [39, 40]. In comparison to BNNs, TNNs achieve an accuracy gain but require double the model size. Therefore, TNNs represent a trade-off between accuracy and model size [40]. Nevertheless, ternary quantization requires retraining to achieve competitive accuracy. In some cases, retrained TNNs surpasses the full-precision accuracy [39, 40].

Zhu et al. introduce a tenary $\{-W_l^n, 0, +W_l^p\}$ quantization method which outperformes the full precision model of ResNet-32,44,56 on CIFAR-10 dataset by 0.04%, 0.16%, 0.36%, and AlexNet on ImageNet by 0.3% [40].

He et al. achieves an accuracy loss of ∼3.9%, 2.52%, 2.16% in comparison to full precision using a $\{-1, 0, +1\}$ ternarized ResNet-18/34/50 on ImageNet dataset.

Log2 quantization

The optimal non-linear quantization strategy depends on the distribution of the weights. Typically, training uses an L_2 regularization, which forces gradients to favor weights close to 0. That results in a normal distribution of the weights of a layer around the mean value zero [41]. Therefore, an alternative for arbitrary non-linear quantization is to use logarithmic quantization since it provides a higher resolution for small weights. In digital systems, log2 is optimal since it allows to transform multiplications into shifts [2, 41]. Compared to linear quantization, Log2 quantization achieves higher accuracy for low-resolution weights of 4 bits and less. Additionally, it enables to replace area intensive multipliers with shifters that allow a higher performance density [41].

2.5 Algorithmic optimization

Algorithmic optimization applies computational transformation or vectorization of data to reduce the number of performed arithmetic operations and memory accesses [2, 17, 42].

2.5.1 General matrix multiplication

Figure 2.7 visualizes the principle of General matrix multiplication (GEMM). It intends to increase the utilization of SIMD structures and to minimize memory accesses for vector-matrix multiplications by combining multiplications with the same weight into a vector. The benefit of GEMM is that a single weight is fetched only once from external memory. GEMM implementations are most efficient for large feature map tensors. That is the case when an entire batch of input data is computed, which is called batching. Figure 2.7 visualizes that the reuse of a weight matrix increases by a factor of N when using N batches. Especially for FC-layers, batching increases the utilization of SIMD processors and reuse of weights [42]. However, batching increases the throughput for the price of higher latency. Therefore, for real-time

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

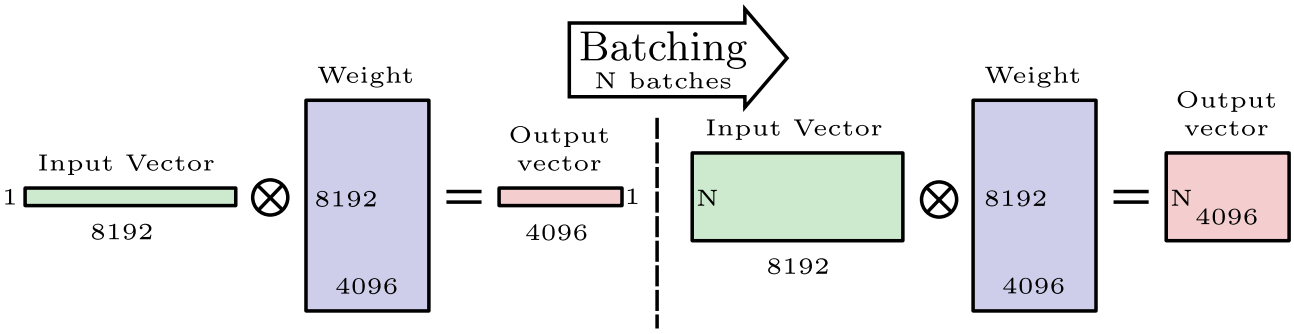


Figure 2.7: Batching converts a vector-matrix multiplication into a matrix-matrix multiplication. It demonstrates that the reuse of the weight matrix increases by a factor of N when batching is used. Figure adopted from [10, Fig. 8].

applications, the use of batching is limited by the latency requirements [42].

Neural network hardware accelerators, widely use GEMM since it increases the efficiency of SIMD and SIMT architectures for matrix multiplications [2, 17, 43, 44].

2.5.2 Winograd transformation

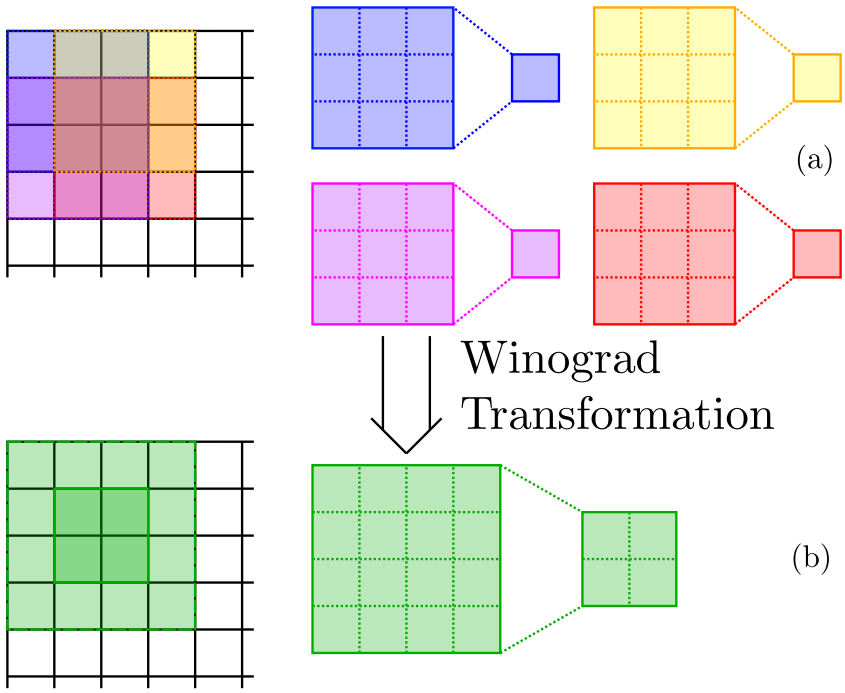


Figure 2.8: Comparison of the computation number required for computing a 2x2 output using four 3x3 convolutions (a) and the computation number required for computing the same 2x2 output using Winograd-Transformation (b). Figure adopted from [10, Fig. 9].

Winograd Transformation is named after its founder Winograd. It reduces the number of multiplication required by finite impulse response (FIR) filters. It transforms overlapping filter kernels into non-overlapping kernels [45]. CNNs use similar moving filters as FIR filter do. Therefore it is possible to adopt the method for CNNs. Figure 2.8 visualizes that by transforming overlapping kernels into non-overlapping kernels, the arithmetic complexity of CNN reduces by a factor up to 4 [46]. It shows that computing a 2×2 output matrix requires 36 multiplications, compared to 16 multiplications needed for the same 2×2 output matrix using Winograd-Transformation. The efficiency increases with the kernel size since more elements are overlapping [46].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

The standard algorithm introduced by Winograd for a $F(2,3)$ uses $2 \cdot 3 = 6$ multiplications [45, 46]:

$$F(2,3) = \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix} = \begin{bmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 - m_4 \end{bmatrix} \tag{2.5}$$

with:

$$\begin{aligned} m_1 &= (d_0 - d_2)g_0 \\ m_2 &= (d_1 + d_2)\frac{g_0 + g_1 + g_2}{2} \\ m_3 &= (d_2 - d_1)\frac{g_0 - g_1 + g_2}{2} \\ m_4 &= (d_1 - d_3)g_2 \end{aligned} \tag{2.6}$$

Equation (2.6) shows that the Winograd algorithm uses 4 multiplications, therefore it is minimal for $\mu(F(2,3)) = 2 + 3 - 1 = 4$. Additionally, it requires four additions involving the data d and three additions and two multiplications involving the filter parameter g. However, these computations can be pre-calculated. That algorithm can also be written in matrix form [46] with $\odot$ indicating an element-wise product:

$$Y = A^T[(Gg) \odot (B^T d)] \tag{2.7}$$

The matrices for $F(2,3)$ are as follows [46]:

$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \tag{2.8}$$

$$G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix} \tag{2.9}$$

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix} \tag{2.10}$$

$$g = \begin{bmatrix} g_0 & g_1 & g_2 \end{bmatrix}^T \tag{2.11}$$

$$d = \begin{bmatrix} d_0 & d_1 & d_2 & d_3 \end{bmatrix}^T \tag{2.12}$$

Equation (2.13) transforms Equation (2.7) into 2D for $F(m \times m, r \times r)$ as follows [46]:

$$Y = A^T[(GgG^T) \odot (B^T dB)]A \tag{2.13}$$

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

with g is a 3×3 filter and d is a $(m + r - 1) \times (m + r - 1) = 4 \times 4$ image tile [46]:

$$g = \begin{bmatrix} g_{11} & g_{12} & g_{13} \\ g_{21} & g_{22} & g_{23} \\ g_{31} & g_{32} & g_{33} \end{bmatrix}$$
$$d = \begin{bmatrix} d_{11} & d_{12} & d_{13} & d_{14} \\ d_{21} & d_{22} & d_{23} & d_{24} \\ d_{31} & d_{32} & d_{33} & d_{34} \\ d_{41} & d_{42} & d_{43} & d_{44} \end{bmatrix} \tag{2.14}$$

It is necessary to perform the additions and subtractions to calculate the new 4×4 weight matrix (GgG^T) only once since they are constant for each layer. However, the new 4×4 matrix $(B^T dB)$ has to be computed before the multiplications. That leads to $16 \cdot 3 = 48$ additions or subtraction required during run-time for each 2×2 output matrix.

A reverse transformation generates the results m from e. That leads to the following additions and subtractions with $e = (GgG^T) \odot (B^T dB)$:

$$\begin{aligned} m_{0,0} &= e_{0,0} + e_{0,1} + e_{0,2} + e_{1,0} + e_{1,1} + e_{1,2} + e_{2,0} + e_{2,1} + e_{2,2} \\ m_{0,1} &= e_{0,1} - e_{0,2} - e_{0,3} + e_{1,1} - e_{1,2} - e_{1,3} + e_{2,1} - e_{2,2} - e_{2,3} \\ m_{1,0} &= e_{1,0} + e_{1,1} + e_{1,2} - e_{2,0} - e_{2,1} - e_{2,2} - e_{3,0} - e_{3,1} - e_{3,2} \\ m_{1,1} &= e_{1,1} - e_{1,2} - e_{1,3} - e_{2,1} + e_{2,2} + e_{2,3} - e_{3,1} + e_{3,2} + e_{3,3} \end{aligned} \tag{2.15}$$

Equation (2.15) requires nine additions or subtractions for computing a single output pixel, which is equal to the standard approach and therefore doesn't influence the performance.
That can be summarized as follows: $F(2 \times 2, 3 \times 3)$ uses $4 \times 4 = 16$ multiplications and 16 additions, compared to that a common algorithm uses $2 \cdot 2 \cdot 3 \cdot 3 = 36$ multiplications, which reduces the amount of required multiplications by a factor of 2.25 [46].

2.5.3 Fast fourier transformation

Fast Fourier transformation (FFT) transforms signals from image to time domain and vice versa. Convolutions in the image domain correspond to multiplications in the time domain and vice versa. Figure 2.9 visualizes the transformation of a convolution operation into an element-wise-multiplication using FFT [2, 17]. It shows that a 2D FFT is computed by calculating first a 1D FFT of every row followed by a 1D FFT of every column [42]. FFT reduces the arithmetic complexity from $O(N_K^2 N_O^2) to O(N_O^2 log_2(N_O))$ with $N_K \times N_K$ indicates the kernel size and $N_O \times N_O$ the output size [2, 17]. The reduction of arithmetic complexity using FFT decrease with the size of the kernel [2]. It is difficult to combine FFT with compact models, pruning or tensor decomposition, which often offers higher profits [2].
Since modern CNNs mainly use small kernel sizes FFT has lost importance for accelerating neural networks.

2.6 Accelerating nested loops

From an algorithmic point of view, matrix computations can be seen as nested loops. The purpose of hardware accelerators is to parallelize the loop iterations. There are three approaches for parallelizing these loops: Loop reordering,

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

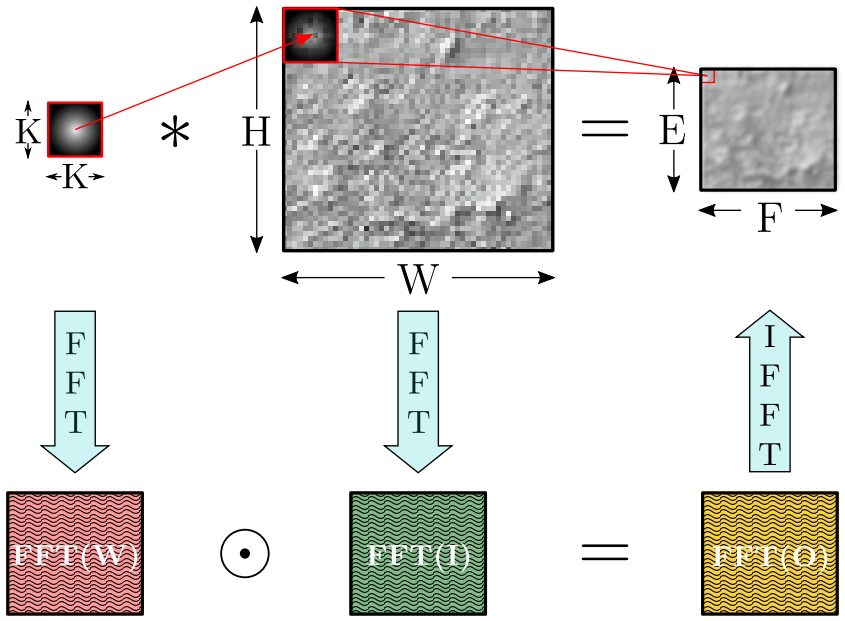


Figure 2.9: Fast Fourier Transformation used to accelerate CNNs. Applying FFT to filters and input feature-maps enables the transformation of a convolution operation into an element-wise multiplication. Figure adopted from [10, Fig. 10].

unrolling, and pipelining. Loop tiling, on the other hand, deals with efficient memory allocation [42].

2.6.1 Loop reordering

It is also known as loop interchange [21]. Loop reordering aims to rearrange the processing of nested loops to avoid redundant memory accesses and to maximize cache usage efficiency [42]. That can lead to a significant performance gain [10, 21].

2.6.2 Loop unrolling

Independent loops can execute simultaneously using parallel hardware. The goal is to decrease the computation time by increasing the hardware utilization [21, 42]. Loop unrolling defines the optimal PE structure [10, 47].

2.6.3 Loop pipelining

Loop pipelining divides nested loops into sequential steps, which can be executed in parallel [42]. The possibility of pipelining depends on the architecture of the hardware accelerator [10].

2.6.4 Loop tiling

Loop tiling partitions the cache into tiles. These tiles are loaded as a chunk from the external memory. That avoids the data in the cache becoming supersede before usage [42]. In terms of hardware accelerators, loop tiling deals with the efficient usage of available on-chip memory to increase the locality of data [47].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Chapter 3

CNN Based Object Detection

CNNs allow a reliable classification of images [3]. For example, CNNs can reliably determine whether there is a car in a picture or not. However, by using ordinary CNNs, it is not possible to find out the position of that car inside the image. For this reason, methods are developed, which use CNNs ability of reliable classifications for object detection [48].

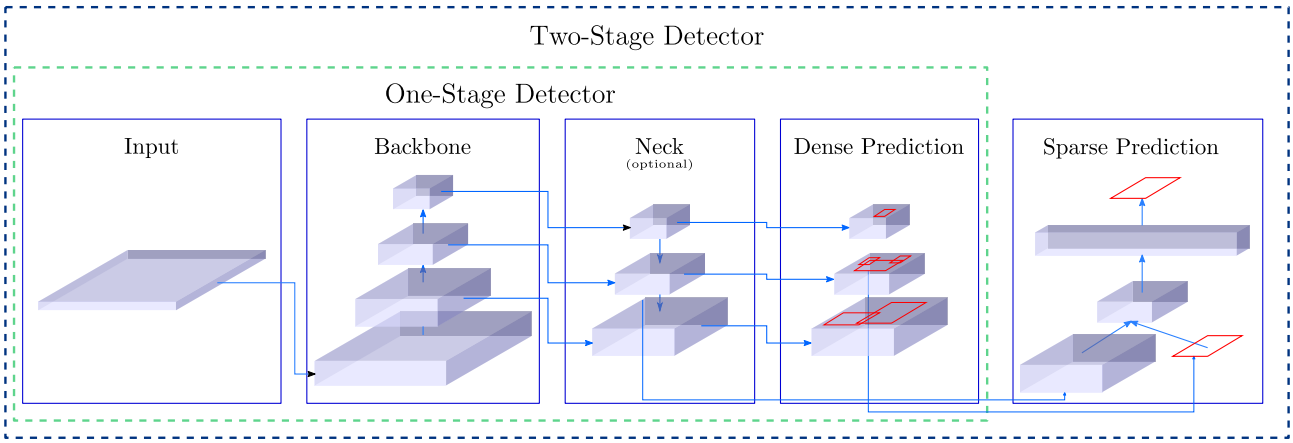


Figure 3.1: Principle of object detectors. Input: Single image, patches or image pyramid. Backbone: Pre-trained classification network. Neck: Collects feature maps from different stages (for scale invariance). Head: Predicts classes and bounding boxes. Dense predict for single stage detectors. Sparse predict for two stage detectors. Adopted from [4, Fig. 2].

Figure 3.1 visualizes the basic elements of CNN-based object detectors. The input can either be a single image, an image patch, or an image pyramid. The backbone of an object detector is typically a classification network, which is pre-trained on the ImageNet dataset [4]. That could be, for example, VGG16, ResNet, or Darknet [12, 49, 50]. The head of an object detector predicts based on the feature maps of the CNN the classes and bounding boxes of the detected objects. The implementation of the head differs with two-stage and one-stage detectors. Two-stage detectors use sparse prediction, which bases on a linear support vector machine. One stage detectors use a dense prediction, which predicts classes and bounding boxes directly based on the feature maps of the backbone CNN or the neck. The neck consists of some layers inserted between the backbone and the head. The function of the neck is to collect feature maps of different layers in a bottom-up path. Using a neck improves the scale invariance by observing feature maps width different resolutions

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

together [4].

3.1 Mean average precision

Defining a measure for accuracy for object detection tasks is complicated since the resulting bounding boxes are not unambiguous correct or false if they are not perfectly overlapping with the ground truth. For this reason, the COCO dataset uses the mean average precision mAP for a defined intersection over union (IoU) [51]. Bounding boxes with an IoU higher than the given IoU level are considered as correct detections. For COCO, there are three metrics. $AP^{50:95}$ is the average of APs for 10 IoU levels from 0.5 to 0.95. AP^{50} uses an IoU level of 0.5, and AP^{75} uses an IoU level of 0.75. The IoU indicates the overlapping area between two bounding boxes and is defined as follows [51]:

$$IoU = \frac{\text{area of overlap}}{\text{area of union}} \tag{3.1}$$

Equation (3.2) shows how to compute the precision P of a result. It indicates the percentage of correct detections. In contrast to precision, the recall R is the ratio of correctly detected objects to available objects in the ground-truth image. Equation (3.3) shows how to compute the recall R of a result. TP is the number of true-positive detections, FP is the number of false-positive detections, and FN is the number of false-negative detections.

$$P = \frac{TP}{TP + FP} \tag{3.2}$$

$$R = \frac{TP}{TP + FN} \tag{3.3}$$

However, there is a trade-off between precision and recall since increasing the confidence threshold increases precision and decreases recall. The interpolated average precision (AP) resolves the dependency of precision and recall on the confidence threshold. For this purpose, it creates a precision-recall curve by adjusting the confidence threshold to get eleven recall levels from 0 to 1. The average precision is the mean of the precisions P_r for these recall levels [51]:

$$AP = \frac{1}{11} \cdot (P_r(0) + P_r(0.1) + ... + P_r(1.0)) = \frac{1}{11} \sum_{n \in (0.0,...,1.0)} P_r(n). \tag{3.4}$$

Equation (3.5) shows the computation of the mean average precision mAP. It is determined by computing the mean value of the average precision of each object class.

$$mAP = \frac{1}{N} \sum_{i=1}^{N} AP_i \tag{3.5}$$

3.2 R-CNN

It represents the first approach of using CNN-features for object detection. R-CNN stands for regions with CNN features. It is a two-stage detector, which uses in the first stage a CNN that generates a feature vector for each region followed by a linear support vector machine in the second stage. Compared to blockwise oriented histogram-based object detectors like SIFT and HOG, R-CNN gives a 30% improvement over the best results on PASCAL VOC 2012 [48]. Some modifications of the region proposal algorithms improve the computation speed of R-CNN. Faster R-CNN manages to accelerate

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

the proposal network and achieves a frame rate of 5 fps [49]. Using ResNet as the backbone of Faster R-CNN achieves an mAP accuracy of 39.8% on the COCO dataset.

3.3 SSD

Single shot multibox detectors (SSDs) are single-stage detectors, which use a discretization of the output space of bounding boxes into a set of default boxes over different aspect ratios and scales per feature map location [52]. To better match the object shape, the network generates scores for the presence of each object category at prediction time in each default box. Each object prediction uses feature maps with different resolutions, which allow handling various object sizes. The structure of SSD is comparable simple, which brings the benefit that it is easy to train and straightforward to integrate into a system [52].
A mobile device-friendly variant of SSD is SSDLite, which uses MobileNet as the backbone. SSDLite uses mainly bottleneck and inverse bottleneck layers, which reduces the parameter number and the computational costs [27, 53]. Comparing YOLOv2 and SSDLite, SSDLite requires less than a tenth of the parameter number and MAC-operations while achieving a comparable accuracy [53].

3.4 YOLO

You only look ones (YOLO) is a single-stage detector intended for real-time object detection, which uses spatially separated bounding boxes and associated class probabilities [54]. A single neural network predicts bounding boxes and class probabilities directly from the image in a single evaluation. Using a single network for the entire detection has the advantage that it can be end-to-end optimized for detection performance [54]. The backbone of YOLO is the darknet. In the original form darknet mainly uses 3×3 convolutions in combination with 1×1 convolutions [54]. In YOLOv3 additionally, residual layers are added, which helps to improve the accuracy [12]. YOLOv4 adds Cross-Stage-Partial-connections, which further increase the accuracy [4].

3.5 RetinaNet

RetinaNet uses ResNet for feature extraction in combination with a feature pyramid net for detection. It introduces a focal loss to address the problem of foreground and background objects. RetinaNet requires a comparable high computational effort, which difficult the use of RetinaNet as a real-time object detector in embedded devices [50].

3.6 Comparision of available object detecors

Table 3.1 shows an overview of existing CNN-based object detection algorithms. The frame rates of the different object detection algorithms are gathered from publications which are using ether Maxwell, Pascal or Volta architecture based Nvidia GPUs with a batch size of 1 [4]. However, the hardware platform and batch sizes influence the achievable frame rate. For this reason, the frame rates in Table 3.1 are only hardly comparable.
Table 3.1 shows that increasing the input image size results in a higher mean average precision. However, increasing the image size comes with the price of a higher computational effort, which results in a lower frame rate. When comparing the object detectors in Table 3.1, YOLOv4 has the best accuracy and a relatively high frame rate. A special approach is

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Model	Backbone	Size	mAP	mAP^{50}	fps^{a}	BFLOPS	Parameter	Source
SSD300	VGG16	300	23.2	43.1	43^{b}	70.4	36.1M	[53]
SSD512	VGG16	512	26.8	48.5	22^{b}	199	36.1M	[53]
SSDLite-V3	MobileNet-v3	224	22.0	N/A	N/A^{e}	1.02	3.22M	[27]
RetinaNet	ResNet-101	500	34.4	53.1	11.1^{b}	N/A	N/A	[50]
RetinaNet	ResNet-101	800	37.8	57.5	5.1^{b}	N/A	N/A	[50]
YOLOv3-tiny	Darknet19	416	16.6	33.1	386^{d}	4.75	8.85M	[55]
YOLOv3	Darknet-53	320	28.2	51.5	45^{b}	38.97	N/A	[4, 12]
YOLOv3	Darknet-53	416	31.0	55.3	35^{b}	65.86	N/A	[4, 12]
YOLOv3	Darknet-53	608	33.0	57.9	20^{b}	140.69	N/A	[4, 12]
YOLOv3-SPP	Darknet-53	608	36.2	60.6	20^{b}	141.45	62.6M	[4, 12, 56]
YOLOv4-tiny	CSPDarknet19	416	21.7	40.2	371^{d}	6.9	N/A	[55]
YOLOv4	CSPDarknet-53	416	41.2	62.8	38^{b}/96^{c}/54^{d}	34.33	27.6M	[4]
YOLOv4	CSPDarknet-53	512	43.0	64.9	31^{b}/83^{c}/43^{d}	52	27.6M	[4]
YOLOv4	CSPDarknet-53	608	43.5	65.7	23^{b}/62^{c}/33^{d}	73.32	27.6M	[4]
EfficientDet-D0	EfficientNet-B0	512	33.8	512	62.5^{c}	2.5	3.9M	[57]
EfficientDet-D1	EfficientNet-B1	640	39.6	640	50^{c}	6.1	6.6M	[57]
EfficientDet-D2	EfficientNet-B2	768	43.0	768	41.7^{c}	11	8.1M	[57]
EfficientDet-D3	EfficientNet-B3	896	45.8	896	23.8^{c}	25	12M	[57]

^{a} Depends on the used hardware and batch size. Batch size: 1. All fps values from [4, Tab 8.].
^{b} GPU architecture: Maxwell
^{c} GPU architecture: Volta
^{d} GPU architecture: Pascal
^{e} No comparable value is available since it is designed for using CPUs.
mAP and mAP^{50} are determined using the COCO dataset.

Table 3.1: Comparison of most common object detectors.

SSDLite, which uses MobileNet as the backbone. It requires only a relatively small computational effort and memory size. That enables achieving real-time capable performance using SSDLite on CPU. That ability comes from the use of bottleneck and inverse bottleneck layers.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Chapter 4

Neural Network Hardware Accelerator

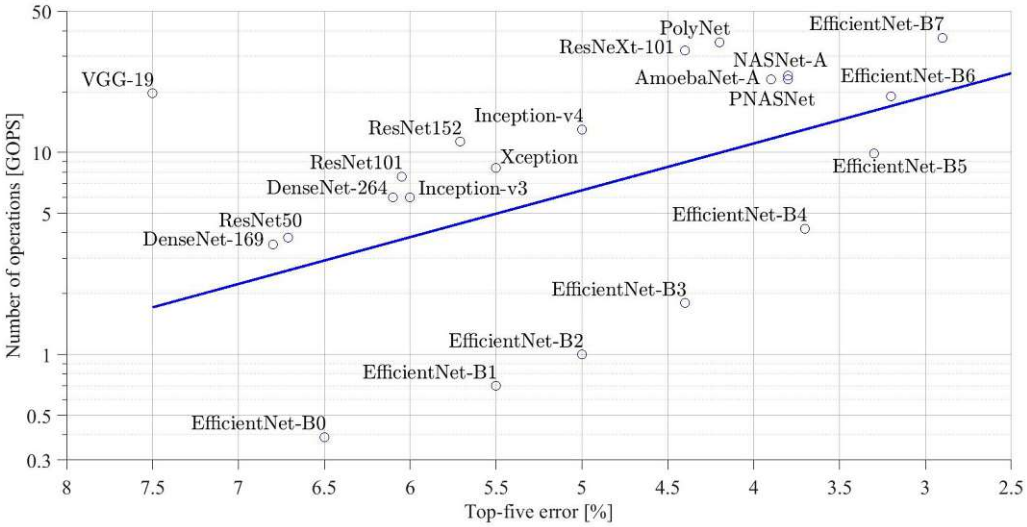
Neural network hardware accelerators are pieces of hardware optimized for the computation of neural networks. However, why do neural networks and specially DNNs required specialized hardware? The reason is that inference and training a DNN, requires an enormous computational effort and gigabytes of data [2, 3]. In addition the trend at DNNs is that the network size is growing exponentially, which leads to an exponentially increasing computational effort and required memory size. That produces an increasing gap between computation power of the hardware and the requirements of the DNNs [3].

Figure 4.1a shows the required number of operations for structural optimized neural networks versus the top-five error using the ImageNet dataset. The Y-axis shows the required number of operations and uses a logarithmic scale. The X-axis shows the top five errors and uses a linear scale. The blue line in Figure 4.1a visualizes that a linear increase of the accuracy leads to an exponential increase of the computational effort. Figure 4.1b visualizes the improvement of the performance density of neural network hardware accelerators. Comparing Figure 4.1a and Figure 4.1b concludes that the number of operations required to compute the inference increases faster than the hardware performance can follow. That leads to a gap between the required hardware performance and the actual available hardware performance [3]. Figure 4.1b shows that the performance density of GPUs using 32-bit floating-point operations depends mainly on the technology feature size[1] of the manufactured die. Figure 4.1b visualizes the technology feature size by the size of the diamonds. Comparing the performance density of GPUs using float32 (yellow) and int8 (green) shows that decreasing the bit width of the arithmetic operations leads to an increased performance density. That is one reason why ASICs can reach a higher performance density compared to general-purpose GPUs [3]. For this reason, modern GPUs implement specialized fixed-point arithmetic units, which increases the performance density and power efficiency of GPUs [58, 59]. However, GPUs still require additional 32-bit floating-point units for use as graphics cards. Nevertheless, GPUs exhibit a higher throughput compared to FPGAs and most ASICs [2, 3].

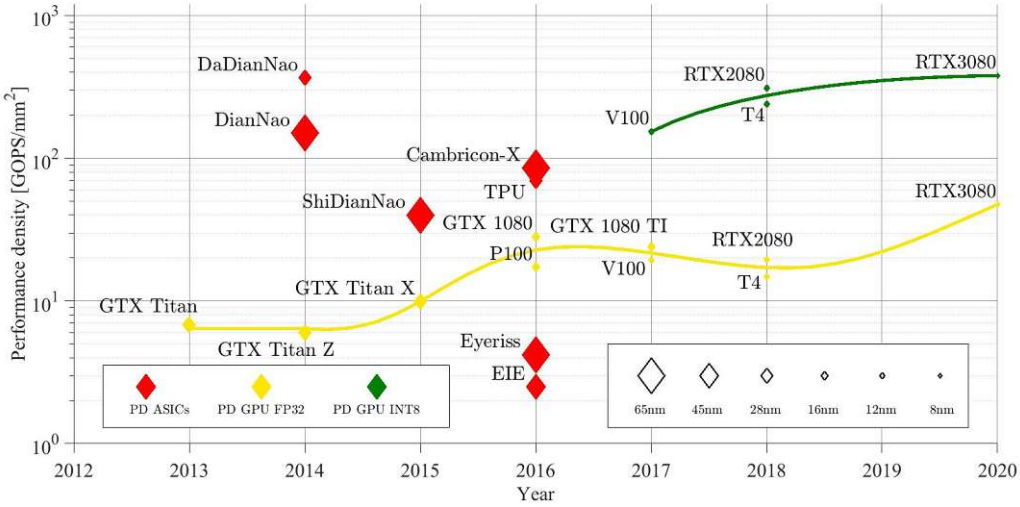
The performance and energy efficiency of hardware accelerators is tied to the technology feature size of the respective manufacturing technology. However, the feature size of a die can only be decreased to a certain point due to Moore's law end. Therefore, increasing the performance density at the same speed as the network parameter number is not possible [3].

[1] In this context, feature size refers to the technology used to manufacture the integrated circuit (IC) and not on the number of feature maps in a CNN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN



(a)



(b)

Figure 4.1: Gap between required number of operations and performance density. (a) number of operations versus top-five error rate for leading DNN designs from ImageNet classification competition. Data from: [14, 16]. (b) performance density (PD) of leading GPU and ASIC platforms. To catch up with the required number of operations, simply increasing the chip area is not feasible. Data from [3, 32–36, 58–61], Adopted from [10, Fig. 5]

There are various neural network hardware accelerator platforms, such as GPUs, ASICs, and FPGAs. Each of these platforms offers certain advantages and disadvantages. This chapter deals with the different hardware accelerator platforms and compares the various implementations.

4.1 Graphics processing unit

GPUs enabled a great breakthrough of DNNs in the field of computer vision. GPUs offer massive parallelization of MAC operations required by DNNs such as AlexNet [11]. Therefore, GPUs are the most widely used hardware accelerator used by DNNs architects [2, 10, 17, 43].

Compared to ASICs and FPGAs, GPUs provide high throughput and are easy to use. However, recent innovations in the structure of neural networks, such as sparsity, BNNs, and TNNs result in a weak algorithmic efficiency using GPUs. These structures can be handled more efficiently using FPGAs or ASICs [3, 10, 43].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

4.1.1 Frameworks

Many frameworks exist for GPUs that offer a high-level API for creating, training, and testing neural networks. Some examples are Caffe, Torch, Theano, Caffe2, PyTorch, TensorFlow, MXNet, CoreML, CNTK, and TensorRT [62]. The simplification of the network design process and the support of training algorithms due to the floating-point computation contribute to the fact that most DNNs architects, which have won the ImageNet competition, use GPUs to accelerate the computations [3, 11, 14, 15, 17, 63, 64].

4.1.2 Architecture

GPUs consist of many floating-point arithmetic units for vector processing in combination with a high bandwidth memory. That is essential for DNNs, since most DNNs bases on dense floating-point general matrix multiplications (GEMMs) [2, 43]. Figure 4.2 visualizes an example of a temporal SIMD/SIMT structure used by GPUs. Temporal SIMD architectures utilize a centralized control for a large number of arithmetic and logic units (ALUs) that share registers and memory. In temporal architectures, each ALU can fetch data from memory and load data back to memory. However, a single ALUs in a GPU cannot communicate with each other directly. Therefore, ALUs exchange data using shared memory. That bottleneck increases the memory traffic, which is power intensive and decreases the possible throughput [2]. Compared to that, spatial structures implemented in FPGAs and ASICs avoid this bottleneck [2].

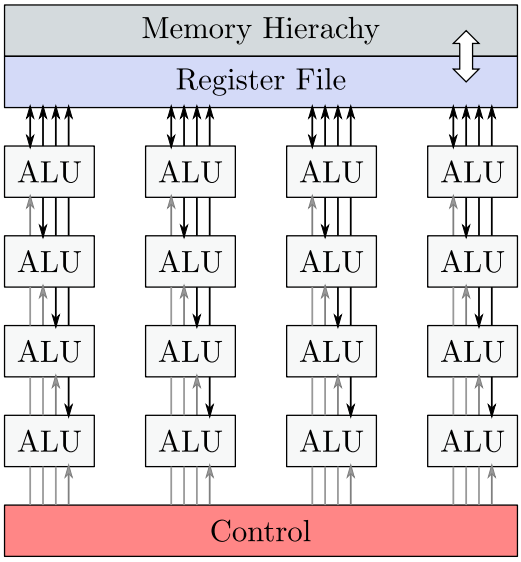


Figure 4.2: Temporal SIMD/SIMT architecture. Adopted from [10, Fig. 11].

4.1.3 Embedded GPUs

Embedded GPU-based hardware accelerators provide an easy to use, cost-efficient platform for fast development since many popular frameworks for neural networks are also suitable for embedded GPUs. Platforms like NVIDIA Jetson combine an ARM CPU with a small power-optimized GPU, for parallel image processing [10, 29].

4.1.4 Overview

Table 4.1 presents an overview of existing Nvidia GPUs. It shows that the introduction of tensor processing units, specially designed for AI, lead to enormous performance gains [10]. Besides, smaller data types are supported, which offers an additional performance gain.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Name	Area [mm^2]	feature size [nm]	Quanti-zation	Bit width	Tensor unit	Throughput [TOPS][(a)]	Freq. [MHz]	Power [W]	$[\frac{GOPS}{mm^2}]^{(b)}$
V100[1] [61]	815	12	float	64		7.8	1530	300	9.57
			float	32		15.7	1530	300	19.26
			mixed	32-8	X	125	1530	300	153.37
T4[1] [58]	545	12	float	32		8.1	1590	70	14.81
			float	16	X	65	1590	70	119.26
			fixed	8	X	130	1590	70	238.53
			fixed	4	X	260	1590	70	477.06
RTX 2080[2] [58]	545	12	float	32		10.6	1710	225	19.45
			float	16	X	84.8	1710	225	155.6
			fixed	8	X	169.6	1710	225	311.19
			fixed	4	X	322.2	1710	225	591.2
RTX 3080[2] [59]	628.4	8	float	32		29.8	1710	320	47.42
			float	16	X	119	1710	320	189.3
			fixed	8	X	238	1710	320	378.7
			fixed	4	X	476	1710	320	757.47
MX250[2]	N/A	14	float	32		N/A	1519	25	N/A
TX 1[3] [65]	N/A	20	float	32		0.512	998	15	N/A
			float	16		1.024	998	15	N/A
AGX Xavier[4] [66]	N/A	12	float	16	X	11	1370	30	N/A
			int	8	X	22	1370	30	N/A

(a) The fictitious peak performance is used due to a lack of comparable data

(b) Performance density

NVIDIA [1]Tesla, [2]GeForce, [3]Jetson (Tegra), [4]Jetson (Volta)

Table 4.1: Overview of GPU-based hardware accelerator implementations.

4.2 Application specific integrated circuits

Compared to GPUs and FPGAs, ASICs provide the greatest degree of freedom for designing a neural network hardware accelerator. However, designing an ASIC requires high effort and in-depth knowledge of chip design and neural networks [9, 42]. Due to the great design freedom of ASICs, ASIC-based hardware accelerators can outperform GPUs and FPGAs in both speed, and power efficiency [3, 9]. A disadvantage of designing a neural network hardware accelerator ASICs is a long time to market and high initial costs. That hinders ASICs from keeping up with the rapid progress in the field of deep neural networks [9, 10].

Table 4.2 gives an overview of existing ASIC neural network hardware accelerators. When comparing the listed implementations, DaDianNao achieves the highest performance density and has moderate power consumption. However, the metrics of DaDianNao are given cryptically as a multiple of the GPU performance.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Name	Area [mm^2]	feature size [nm]	Quanti-zation	Bit width	Throughput [GOPS]$^{(a)}$	Frequency [MHz]	Power [mW]	$[\frac{\text{GOPS}}{\text{mm}^2}]^{(b)}$
DaDianNao [60]	4335*	28	fixed	16	1586288*	606	48380*	366*
EIE [36]	40.8	45	fixed	16	102	800	590	2.5
Cambricon-X [35]	6.38	65	fixed	16	544	1000	954	85.26
Eyeriss [33]	16	65	fixed	16	67.2**	200	278	4.2
TPU [34]	331***	28	fixed	8	92000	700	40000	69.48

(a) The fictitious peak performance is used due to a lack of comparable data

(b) Performance density

*For 64 nodes. In paper only stated as a multiple of NVIDIA K20M

**Stated in the paper in GMAC. Therefore the double value is used

***4 dies

Table 4.2: Overview of existing ASIC-based hardware accelerator implementations.

4.2.1 DianNao family

The architecture of DianNao uses global input data, weight, and output data buffers in combination with a neural function unit (NFU) for computations. The NFU consists of 16-bit fixed-point multipliers in the first stage, an adder tree in the second stage, followed by a non-linearity unit. The second stage additionally contains shifters and max operators for pooling layers. The goal of DianNao is to support neural networks of any scale [32, 67]. In DaDianNao, adding additional buffers next to the neurons reduces the data movement. Additionally, the single computational block is split into four identical NFUs. DaDianNao targets high performance [32, 60]. ShiDianNao improves the DianNao structure by using the fact that in CNNs a whole vector shares the same weight. This condition is used to reduce the required DRAM accesses and to reduce the power consumption, therefore [10, 32].

4.2.2 EIE: Efficient inference engine

EIE is designed for operating on sparse networks. That brings the benefit that faster and more power-efficient SRAM replaces power-hungry DRAM memory. The architecture of EIE utilizes a central control unit, which controls a PE array. Each PE computes one slice of the compressed network. The central control unit broadcasts non-zero activation values to the PEs, which use an activation queue, for buffering input data [10, 36].

4.2.3 Eyeriss

The core architecture of Eyeriss is a spatial array with 168 PEs in a 12×14 shape, including a 108-kB GLB, an RLC CODEC, and a ReLU module. Each PE can communicate with the neighboring PEs or the GLB. A PE contains a MAC unit, a local buffer called spads, and a control unit [10, 33].

4.2.4 Tensor processing unit

A matrix multiply unit, which includes 256×256 MACs, is the core element of the tensor processing unit. The multiplication unit has an input bit width of 8, the accumulation unit a bit width of 16. The matrix multiplication unit is followed

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

by 4096, 256-element, 32-bit accumulators. Additionally, a dedicated activation module for applying the non-linearity and a dedicated normalization and pooling module is used. A 24 MiB on-chip buffer is used for storing the results. The weights are loaded directly from DRAM into a specialized weight FIFO [10, 34].

4.3 Field programmable gate arrays

FPGAs offer a cost-efficient method to develop custom hardware solutions since FPGAs provide the possibility to implement reprogrammable logic circuits. The implementation of a task in a logic circuit offers the advantage of high parallelization compared to an implementation using a CPU. That leads to considerable speed advantage and higher power efficiency [68, 69]. Additionally, the implementation of an accelerator in hardware reduces the necessary memory accesses since the intermediate results are passed on directly to the next PE. In addition to configurable logic blocks (CLBs), modern FPGAs implement hundreds to thousands of DSPs and various options for storing data locally. Embedded BRAMs and FIFOs are available to store larger amounts of data. In contrast, LUTs are well suited for the use of distributed RAM for smaller amounts of data. That enables FPGA-based neural network implementations to minimizes the delay due to memory access [10].
For this reason FPGAs offer the possibility to realize a low latency inference in combination with a high energy efficiency ($\sim 10 - 50\mathrm{GOP/s/W}$) [47]. All in all, FPGAs represent a tunable balance between performance and energy consumption [62]. However, greater freedom of design also inevitably leads to a more complex solution and more development effort [10].

4.3.1 Algorithmic optimization in FPGA based hardware accelerators

FPGA-based hardware accelerators widely use the algorithmic optimization methods described in Section 2.5. However, the efficiency of these methods depends on the structure of the hardware accelerator. Hardware structures that support batching, such as single computation engines, are well suited for the use of GEMM. Nevertheless, GEMM is not suited for streaming architectures since multiple batches would require a large tile size of the buffers [10, 47].
Basically FPGAs would be well suited for the use of FFT. However, FFT is more efficient for large kernel sizes and the trend in DNN structures is towards small kernel sizes [2, 10, 12, 14, 15].

4.3.2 Mapping convolutional neural networks on FPGAs

The task of mapping a DNN having millions of parameters and ten to hundreds of layers is a complex multidimensional optimization problem [47]. The goal is to achieve high throughput and low energy consumption. Therefore one goal of mapping CNNs on FPGAs is to minimize off-chip memory accesses since these lead to high energy consumption and latency [2, 47]. Second, a high throughput requires a high hardware utilization and an efficient design. Loop optimization methods such as loop unrolling, loop tiling, loop reordering, and loop pipelining target that task [2].
In low cost FPGAs on-chip memory is not sufficient for the large amounts of data to process DNNs. Therefore, FPGA-based hardware accelerators typically use the following three memory hierarchies: external memory, on-chip buffers (cache), and registers for the individual PEs.
Due to the limited hardware resources, full unrolling of large DNNs is often not possible. Therefore, most hardware

[2] See Section 2.6

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

accelerators buffer intermediate results locally in BRAMs or the external memory. The fewer intermediate results have to be stored in local buffers, and the earlier the hardware accelerator processes the data stored in these on-chip BRAMs, the less off-chip memory accesses are required. Therefore the efficiency increases. Additionally, to optimally utilize the on-chip buffers, suitable tile sizes are required [47].

A difficulty with using FPGAs is the complex design flow to get an efficient implementation of a neural network structure. High level synthesis (HLS) bases on widely used programming languages such as C, C++, and OpenCL. Due to a higher abstraction level, HLS promises to simplify and accelerate the design flow. In addition, it offers the possibility of automating the design flow based on the network structure and the parameters of each layer. That enables neural network designers without any hardware programming experience to create a customized hardware implementation [62]. Venieris et al. provide an overview of existing CNN-to-FPGA tool flows, and features such as performance, arithmetic precision, portability supported NN-models, and optimization are compared [62]. However, abstraction comes with a loss in efficiency [31].

4.3.3 Hardware architectures

Typical automated toolflows use one of following 2 hardware structures: Single computation engines and streaming architectures [62].

Single computation engines

Single computation engines typically implement systolic array structures [62]. Figure 4.3 visualizes a typical systolic array structure. It shows that in systolic arrays, each PE exchanges data with neighboring PEs. That enables systolic arrays to achieve high data rates between the PEs. The PEs of systolic arrays can either consist of MAC units, convolutional PEs, or SIMD units [30, 62]. Figure 4.3c shows a block diagram of a convolutional PE. However, weights, inputs, and outputs are transferred from or to the external memory, which leads to a bottleneck. Adding additional buffers for weights and input data reduces the bottleneck [30]. The benefit of using systolic arrays is that they can process different DNN structures. However, flexibility comes with the price of reduced efficiency. That leads to a high difference in throughput and latency using different DNNs [8, 10, 70].

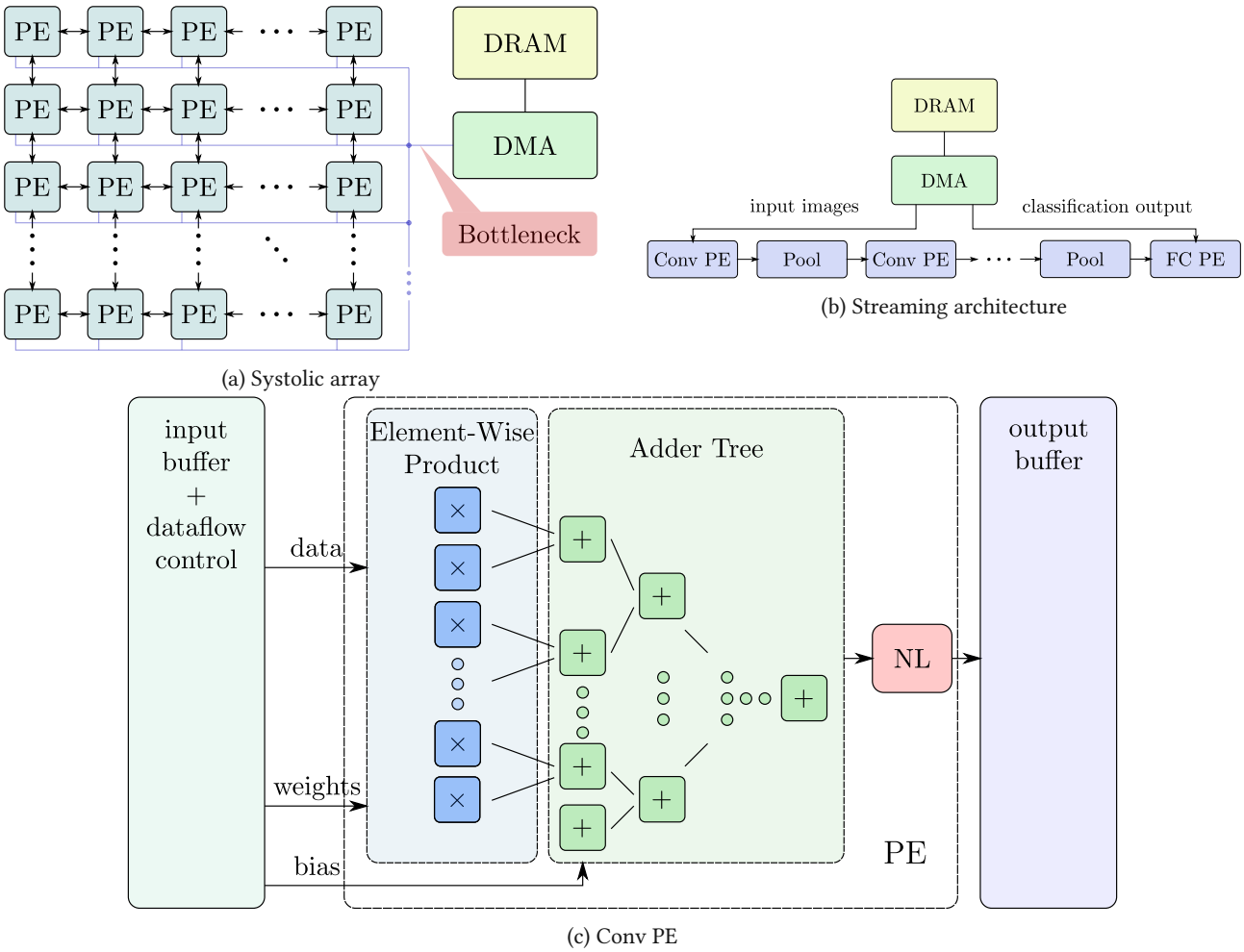
Streaming architectures

Figure 4.3b visualizes the structure of streaming architectures, which implement an independent specialized PE for each layer. The architecture enables pipelining the data through the hardware accelerator. One design goal of streaming architectures is to ensure that the calculations of individual layers take the same amount of time to prevent the PEs from idling. Each layer stores the weights in local buffers, which reduces the global data transfer. That enables streaming architectures to implement a high efficient accelerator. In comparison to single computation engines, streaming architectures are less flexible since they are optimized for a single DNN structure. Additionally, increasing or decreasing the streaming architecture size for optimal utilization of the FPGA resources is difficult [10, 62].

4.3.4 Overview of existing implementations

In 2015 Zhang et al. introduced a 32-bit floating-point single computation engine on a high-end Xilinx FPGA using HLS. The accelerator uses 64 two-level unrolled PEs operating with 100MHz. Each PE has seven inputs for activations

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN



(a) Systolic array

(b) Streaming architecture

(c) Conv PE

Figure 4.3: Typical hardware architectures used in an FPGA-based hardware accelerator. a typical structure of systolic array. PEs are able to communicate with adjacent PEs. Weights, input and output data is transferred using DMA. b typical structure of a streaming architecture. Each layer is computed by an independent PE. The data is pipelined throw the accelerator. c typical structure of convolutional processing element. Consist of N multipliers depending on the kernel size, followed by an adder tree to sum up the intermediate results and a non-linearity element. Adopted from [10, Fig. 12].

and weights and one bias input. Additionally, each PE has input and output buffers. Loop tiling determines the size of on-chip buffers. A MicroBlaze CPU controls the data flow of the hardware accelerator. The system has a power consumption of 18.61W. Due to the high precision PEs, no accuracy loss occurs. However, the high precision results in a low throughput of 61.62GOPS [71]. For this reason, that design has compared to others a poor implementations efficiency [10].

In 2016 Zhang et al. created a hardware-software co-design library called Caffeine, which reduces the design effort for creating a hardware accelerator based on Caffe. It has similar performance metrics compared to the previous version. However, in this version 16 bit fixed point quantization is available, which increases the performance by a factor of 3.8 [10, 28].

Also, in 2016 Qiu et al. presented a 16-bit fixed-point single computation engine specialized for convolution operations. This structure brings the benefit of more efficient convolutions computations. However, due to the non-optimal data flow for FC layers, they can only be calculated with low-performance [19]. The overall throughput is similar to the 16-bit fixed-point implementation introduced by Zhang et al. [19, 28]. Additionally, Qiu et al. estimate the throughput

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

of a 4/8 bit Quantization to be 2.6 × higher with a reasonable low accuracy loss of 0.4% [10, 19].

Wei et al. created an automated systolic array architecture synthesis in 2017. The unique thing in this implementation is that each PE contains a SIMD vector accumulator. The systolic array structure aims to maximize the on-chip data reuse. For this reason, additional buffers for input, output, and weights are placed at the systolic array edges [30]. This implementation achieves compared to Zhang et al. a very high throughput of 461GOPS using a 32-bit floating-point, and a 16-bit fixed-point implementation [28, 30, 71]. However, a comparison of the implementations is difficult since the implementations use different FPGA vendors that utilize different DSP structures [10].

Aydonat et al. use 16-bit floating-point operations for accumulations and 16 bit fixed point operations for multiplications to reduce the overhead from using FP32 DSPs. The hardware accelerator structure uses serial PEs, optimized for computing Winograd transformed feature maps [72]. This design outperforms the 16-bit implementation of Wei et al. by a factor of 1.18 in terms of throughput [10, 30, 72].

Liang et al. introduced in 2018 a BNN for FPGA. Bit-level XNOR and shift operations replace costly multiplications due to the binarization of both weights and activation. That enables a massive boost in throughput. However, an extensive drop in accuracy occurs for complex DNNs such as AlexNet. The use of full-precision computations in the last layer counters this effect. Nevertheless, an accuracy loss of 14.7% occurs using AlexNet [10, 38].

Chung et al. and Fowers et al. presented the Brainwave neural processing unit (NPU) in 2018. Which is a single thread SIMD instruction set architecture paired with 96000 MACs for vector-matrix and vector-vector operations. The target of the hardware accelerator is maximal performance for cloud-scale real-time applications. The quantization is a 9-bit pseudo floating point quantization, including a sign bit, a 5-bit exponent, and a 2-5 bit mantissa. 128 numbers share a common exponent. This quantization results in an accuracy loss of 1-2%. When performing ResNet-50 on an Arria 10 1150, the brainwave NPU achieves a lower real-time latency (batch size 1) than an Nvidia P40 GPU using int8 quantization [7, 10, 73].

Nguyen et al. present a real-time YOLO object detection streaming architecture on a Xilinx XC7V485. It implements a specialized PE for each layer and pipelines the images through the processing elements. The design minimizes the accelerator footprint by using binary quantized weights and 4 to 6-bit quantization for the activations. That leads to an accuracy loss of 2.5%. This implementation achieves a 1.24× higher frame rate with an 11.5× higher energy efficiency compared to an NVIDIA GTX Titan X GPU using Slim-YOLO-v2 [10, 31].

Table 4.3 provides an overview of the introduced related FPGA-based hardware accelerator implementations. It compares the quantization with their influence on the throughput and utilization efficiency to estimate implementation efficiency. The throughput divided by the available logic elements (LEs) and the throughput divided by the available DSPs is used as a measure for the utilization efficiency because it shows how efficiently the architectures use available resources to generate maximum throughput [62]. The number of DSPs and LEs correlate, which leads to a similar comparison. Historically, a single LE is defined as a four-input LUT, a programmable register, and a carry chain. However, in modern FPGAs LUTs usually have more than four inputs. For this reason, FPGA vendors use an adjustment factor for the conversion into a fictitious number of 4 input LUTs. Nevertheless, a comparison of implementations using different

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Article	Quant-ization	Bit width W	Bit width X	Accuracy loss[(a)]	Throughput [GOPS]	Freq. [MHz]	Power [W]	FPGA Type	FPGA # LE	FPGA # DSP	$\frac{GOPS}{k\,LE}$	$\frac{GOPS}{DSP}$
Zhang et al. [71]	float	32	32	0%	61.62	100	18.61	XC7VX485	485.7k	2800	0.13	0.02
Zhang et al. [28]	float	32	32	0%	96	200	25	XCKU060	726k	2760	0.13	0.03
	fixed	16	16	N/A	365		25				0.5	0.13
Qiu et al. [19]	fixed	16	16	0%	187.8	150	9.63	XC7Z045	350k	900	0.54	0.21
		4	8	0.4%	495**		N/A				1.41	0.55
Hegde et al. [29]	fixed	16	16	N/A	11.5	180	19	XC7Z045	350k	900	0.03	0.01
Wei et al. [30]	float	32	32	0%	461	221.65	N/A	Arria 10	1150k	1518	0.4	0.3
	fixed	8	16	2%	1171	231.85	N/A				1.02	0.77
Guo et al. [8]	fixed	16	16	0.06%	187.8	150	9.63	XC7Z045	350k	900	0.54	0.21
		8	8	0.62%	19.2	100	2	XC7Z020	85k	220	0.23	0.08
Aydonat et al. [72]	float	16	16	0%	1382	45	303	Arria 10	1150k	1518	1.2	0.91
Liang et al. [38]	binary	1*	1*	0.46-14.7%	9396	150	26.2	Stratix V	695k	3926	13.52	2.39
Chung et al. [7]	float	9	9	1-2%	39500	300	125	Stratix 10	2800k	5760	14.1	6.86
Ma et al. [47]	fixed	16	16	2%	348	150	N/A	Stratix V	622k	3926	0.56	0.09
					715	200		Arria 10	1150k	1518	0.62	0.47
Venieris et al. [70]	fixed	16	16	N/A	48.53	125	<5	XC7Z020	85k	220	0.57	0.22
					155.81		<5	XC7Z045	350k	900	0.45	0.17
Nguyen et al. [31]	binary	1*	4-6*	2.5%	1877	200	18.29	XC7VX485	485.7k	2800	3.36	0.67
Wang et al. [74]	fixed	8	8	N/A	5531	300	25	XCZU7EV	504k	1728	3.36	3.2

(a) compared to 32bit floating-point
*Not applied to first and/or last layers.
** Estimated value.

Table 4.3: Overview of existing FPGA-based hardware accelerator implementations.

FPGA-vendors is difficult.

Table 4.3 shows that quantization is crucial for implementation efficiency. However, the efficiency gain comes with an loss in accuracy. Nevertheless, a much higher throughput gain than the accuracy loss can be observed if the design uses a suitable combination of weight and activation bit width. Furthermore, latest implementations show that FPGAs challenge GPUs in real-time computation tasks [31, 73].

4.4 Comparison

The decision of which neural network hardware accelerator is the most suitable depends on the application. For this reason, this section compares the throughput, latency, accuracy, power, and usability separately. Since accuracy, throughput, and energy cost is a trade-off, the aim of the comparison is not to determine which platform is generally the best, but rather which platform suits best for which application [10, 69].

A distinction between throughput and latency is necessary since these parameters can differ considerably depending on the hardware accelerator and batch size. For real-time applications latency is the most crucial measure. Throughput is of secondary interest. The opposite is true for all other applications [10].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

4.4.1 Throughput

Due to the high clock frequency, the large die size, and the SIMD architecture, which provides high parallelism, GPUs can offer up to some Tera FLOP/s throughput [34,38]. However, using specialized PEs and a lower bit width, ASIC-based implementations can achieve higher throughput compared to general-purpose GPUs. However, Figure 4.1b shows that the introduction of tensor processing units in GPUs, which enable the use of a reduced bit width, achieves a performance density comparable to that of ASICs. Comparing Table 4.2 and Table 4.1 shows that GPUs achieve a higher performance density compared to TPU using 8 bit fixed point quantization. Nevertheless, TPU is more power-efficient and uses a larger technology feature size [10].
Compared to GPUs and ASICs, FPGA-based implementation achieves a lower throughput. That comes from the fact that the variable routing of an FPGA limits the bandwidth and is area intensive [10].

4.4.2 Latency

As mentioned before, latency is critical for real-time applications, which are implemented mainly in edge devices. In real-time applications, the ability to batch the input data is limited to meet the real-time requirements [7, 10, 75].
How efficient GPUs process neural networks depends on the size of the matrices and thus on the batch size. For this reason, GPUs lose performance in real-time applications, since the GEMM-based optimization is less efficient using a small batch size [7]. Therefore, ASIC or FPGA-based implementations, which do not rely on GEMM can achieve a better throughput to latency ratio. Chung et al. show that FPGA-based hardware accelerators can outperform an Nvidia Tesla Xp GPUs in real-time applications if fixed-point quantization is used [7]. Besides, note that the higher design freedom of ASICs and FPGAs enables more efficient processing of compact models. However, for a more complex data flow, an additional chip area is necessary [10].

4.4.3 Accuracy

The accuracy of a DNN mainly depends on the neural network structure, the dataset, and the training methods. These parameters are independent of the hardware accelerator. Many neural network hardware accelerators use quantization to increase performance density. However, that influences the accuracy of the required computations and can lead to an accuracy loss. In general, for both floating-point and fixed-point quantization, using a bit width of 16 bits or higher does not lead to an accuracy loss. Otherwise, quantization-aware training can recover the accuracy to minimize the accuracy loss. Retraining can even lead to a performance gain for some not optimized neural network structures such as AlexNet [10, 40].
Table 4.3 gives an overview of the used bit width and the resulting accuracy loss. It shows that a more aggressive quantization method, such as binary weights combined with 6-bit activations, leads to only a minor accuracy loss. That leads to an enormous gain in terms of throughput and latency [10, 31].

4.4.4 Power

Power consumption is a key element for battery-powered embedded systems such as mobile devices, robots, etc. [38]. However, for data centers, power consumption is also of interest since energy and cooling costs make up a significant part of the running costs of a data center [76]. For this reason, FPGAs and ASICs become more attractive for data centers since they offer advantages in the utilization of computation capacity and high energy efficiency [7, 73]. FPGAs

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

usually achieve more than 10 times higher energy efficiency compared to GPUs [38]. However, comparing Table 4.1 and Table 4.3 shows that this is only true if the GPU uses floating-point-32. Table 4.2 visualizes that except for TPU, all other ASIC implementations have a very high power efficiency. However, TPU has a better power efficiency compared to GPUs and high throughput. Therefore, TPU is a favorable solution for data centers. For power-critical embedded devices, ASICs or FPGAs are a feasible choice [10].

4.4.5 Usability

Many frameworks exit for GPUs that offer a high-level API for creating, training, and testing neural networks (See Section 4.1). Using the tensor cores of a GPU requires a C++ library called TensorRT. It includes a parser that enables an import of existing models from the most neural network training APIs. For this reason GPUs provide the best usability compared to ASICs and FPGAs [10].

However, TPU supports the high-level TensorFlow framework, which considerably increases usability. Since the TPU version introduced by Jouppi et al. uses int8 quantization, training is not recommended [34]. Therefore, next-generation TPUs support floating-point operations and training. Since other ASIC implementations do not support high-level APIs, they have poor usability. For FPGAs some frameworks based on HLS and OpenCL exist (See Section 4.3). Nevertheless, more in-depth knowledge of neural networks and hardware design is necessary for implementing an efficient FPGA-based hardware accelerator. Just like ASICs, FPGAs usually do not support training. Compared to ASICs, FPGAs have the advantage that they have a short design cycle, which is crucial in the field of neural networks because progress is fast [10, 68].

4.4.6 Fields of application

General-purpose GPUs are well-suited for the training of neural networks since they provide a high number of floating-point ALUs. However, the SIMD/SIMT architecture of GPUs lead to many DRAM accesses. In combination with the use of power-hungry floating-point computations, the high number of DRAM accesses lead to the high power consumption of GPUs. Therefore, in modern GPUs, additional tensor processing units are added, which provide an improved data flow and a quantization down to int4 [10].

In contradiction to general-purpose GPUs, ASIC-based neural network hardware accelerators are highly optimized for tensor computations. Therefore, ASICs provide both high throughput and high power efficiency. However, creating a fully custom neural network hardware accelerator requires a long design time and an in-depth knowledge of chip design and neural networks. Since the progress in the field of deep learning is fast, the time to market is essential. In addition, a high-level API, which simplifies the implementation of a neural network in the hardware accelerator, is required to enable the uses for all neural network architects [10].

FPGAs offer a programmable logic circuit that enables a highly optimized design with comparable low design effort and initial costs. In addition to that, some frameworks exist based on HLS, which automatically generates an optimized hardware implementation based on the neural network design. Recent advances in sparsity and quantization enabled FPGAs to achieve a throughput comparable to general-purpose GPUs while having a higher power efficiency. In addition, FPGAs offer the possibility to optimize the data flow for reducing the required memory bandwidth for sparse neural network structures. Therefore, FPGAs are a promising candidate for use in real-time edge systems. They also offer the advantage over ASICs in that they are reprogrammable. That is a considerable benefit due to the rapid progress in the field of deep learning [10].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

It can be concluded that FPGAs are a feasible choice for a real-time application with a limited power budget. ASICs are well suited for the use in data centers since the high initial costs and the long time-to-market are acceptable to achieve a higher throughput and energy efficiency. For all other applications, GPUs are the most feasible choice [10].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Chapter 5

Neural Network Hardware Accelerator Design

The goal is to design a hardware accelerator for real-time object detection in a low-cost FPGA device such as a Xilinx Zynq-7020. Since progress in the field of object detection is fast, constantly creating advanced backbone and detector structures, the hardware accelerator should support various object detector designs, to be fit for future implementations. The hardware accelerator architecture is designed as a single computation engine[1], because streaming architectures require dedicated hardware for each operation, which would exceed the available resources for implementing a DNN in a low-cost FPGA device. Moreover, single computation engines provide higher flexibility for different neural network structures. Nevertheless, they suffer from higher data traffic and therefore the throughput is limited by the external memory bandwidth.
That problem is counteracted, by using local buffers in combination with targeted feature map tiling. The goal of feature map tiling is to utilize the caches and PEs as best as possible and, ideally, to load the required data only once from the external memory. The intermediate results are kept in the local cache until the computation of the corresponding output is done.

Figure 5.1 shows the structure of the implemented neural network hardware accelerator. The core of the accelerator consists of 16 PEs, which share the same input data, but use different weights. The PEs are designed to compute a single output channel of a convolutional layer independently of one other, which enables the computation of 16 output channels in parallel. Two different processing units are implemented to investigate the effect of int8 and float8 quantization on the throughput and accuracy. Each PE consists of 12 PUs, whereas dependable of the PU implementation, each PU can do 2 MAC-operations in parallel when using int8 and 3 MAC-operations in parallel when using float8. That lead to a maximum of 24 parallel MAC operations for each int8 PE and 36 MAC operation for each float8 PE. The dataflow control block in combination with the data streamer is responsible for broadcasting the data and the weights to the PEs according to the current operation mode. The collector block fetches the completed feature map tile from the PE caches and stores the results in dedicated output buffers.

[1] See Section 4.3.3

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

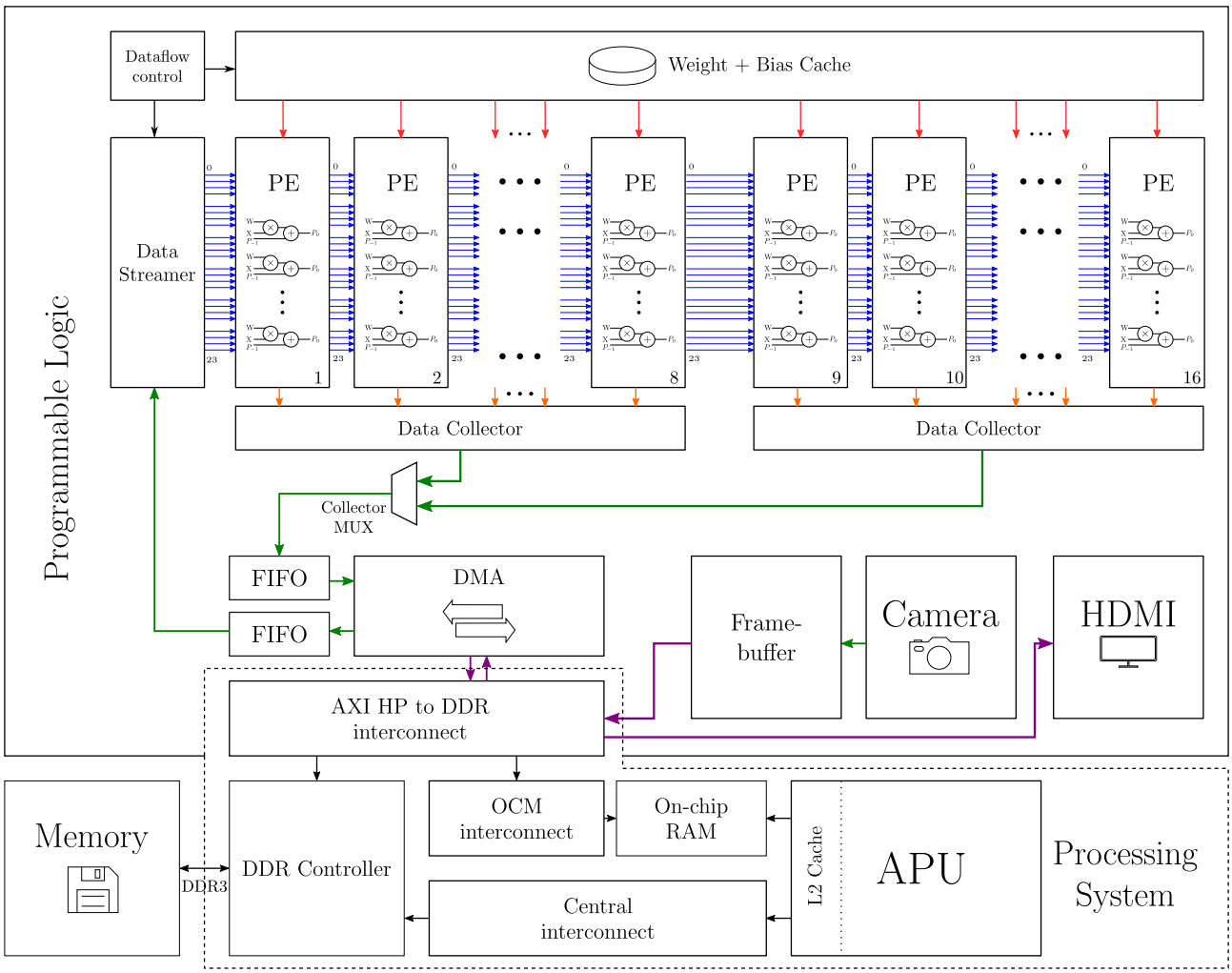


Figure 5.1: Overview of all components used for the neural network hardware accelerator.

The camera module sends the received images to the external memory and not directly to the neural network hardware accelerator. The reason for storing the image in the external memory before using it is that the image size exceeds the local cache size. In addition, the image data could be needed by another application. That could be for example plotting the results to a screen.

5.1 Data streamer

Figure 5.2 visualizes the components of the data streamer, which provides the required data and weight for the PE dependable on the current operation mode. The data streamer receives the commands, layer parameters, and the input data from the external memory via an AXI-direct memory access (DMA) intellectual property (IP) and stores the parameters and input data in dedicated caches. The transfer of a new tile starts with a new-weight command. It includes the length of the weights and biases and the start address for the weight, bias and data streamer caches. It is possible to reuse weights and biases that are already in the memory. That can be done by setting the address in the command to the start address of the reused weight block and the new weights flag is set to zero.

The weight cache has double the size of the bias cache since the number of weights is much higher in comparison to the number of biases in a convolution layer. However, if there are enough BRAM slices left in the design, it is recommended

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

to increase the weight cache to 4 or 8 times the bias cache size.

The feature map command is sent after the bias and weight block. It indicates among other things the tile height and tile width of the new feature map tile and whether the data is sent word aligned line by line or as a continuous data stream. These parameters are required to store the data correctly in the corresponding feature map cache. The line reshape control uses the line buffers to transpose the data, since the systolic array expects the input data to be column by column.

After storing the transposed data into the data streamer cache the dataflow control reads the data from the caches according to the current dataflow[2] required for the used operation mode and stride. The streaming process starts if all PEs are ready.

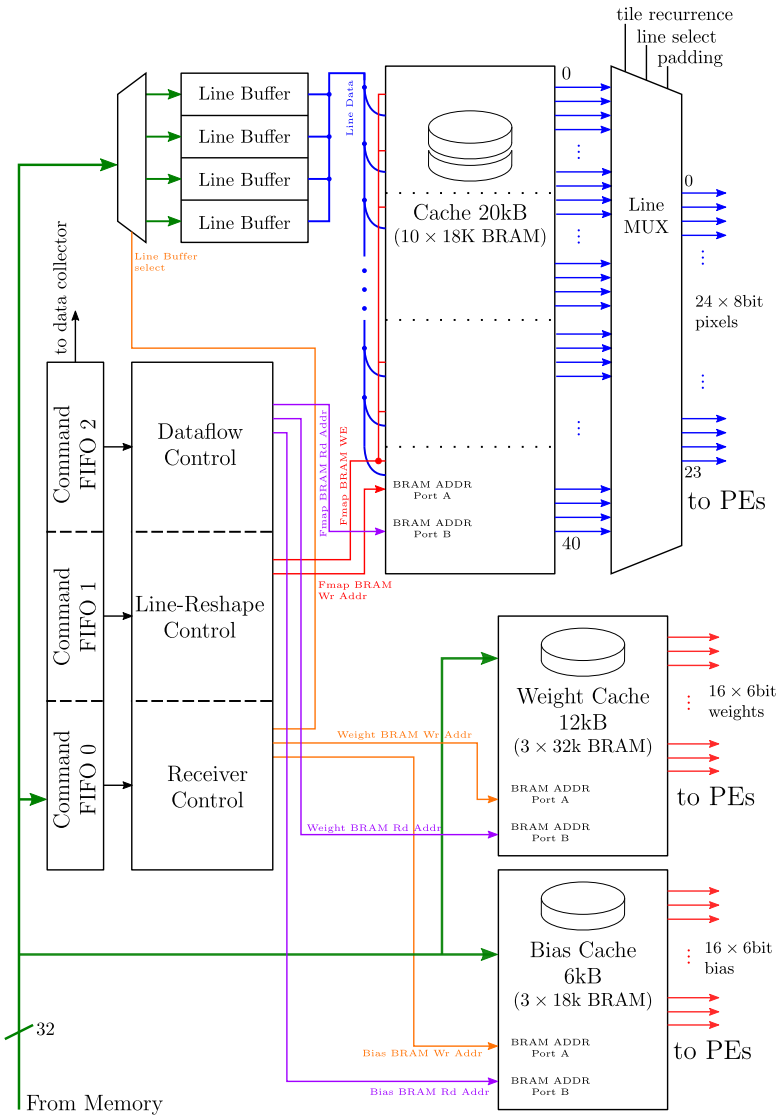


Figure 5.2: Block diagram data streamer. The diagram shows how the data is written to the cache by the receiver and how the dataflow control streams the data to the PEs.

5.1.1 Tile recurrence

One problem with this design is that with tile heights smaller than the maximum tile height, not all PUs are utilized. This effect increases the smaller the tile height is. For example, using the int8 implementation and a tile height of 12, half of the PUs are used. Tile recurrence counteracts this drawback. It splits the systolic array into 2 or 4 "recurrent"

[2] See Section 5.2

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

parts, which allows the computation of 32 output channels for a tile height up to the half of the maximum tile height and 64 output channels for a tile height up to a quarter of the maximum tile height. Tile recurrence divides the feature-map cache into two or four equal parts and clones the input data of the current tile into all recurrent caches. Cloning the data does not need additional time since the structure allows a simultaneous write to all required BRAMs. That enables the otherwise not utilized processing units to do computations for a different output channel with the same input data. However, weights for the additional output channels have to be provided for these PUs. For this reason, additional weight and bias caches are required to provide the supplementary weights and bias. Tile recurrence avoids poor PE utilization for small tile heights.

5.1.2 Padding

The hardware accelerator supports "same" padding for convolutional layers. Dependable of the used kernel size and stride, zero lines or columns are added to the input data tile. The feature map command reserves 4 bits for padding. Each combination of top, bottom, left or right padding is allowed. The line MUX handles top and bottom padding. It toggles the reset signal of the corresponding line to force the line to zero. For this reason, the content of that line is irrelevant and does not have to be overwritten. For left and right padding, an extra zero column is added to the cache at the start or end of its address space. The feature-map cache enables resetting all lines of a single address in a single cycle. For this reason, adding a padding column requires only a single cycle.

5.2 Dataflow control

Figure 5.3 shows a block diagram of the dataflow control and the interaction of the modules. Based on the command the dataflow control generates the control signals for the systolic array, the read and write addresses for the PE caches, and the read addresses for the feature map, the weight, and the bias cache. The Dataflow state machine extracts the information in the command vector and initializes the other modules for the new tile. The dataflow timer is the heart of the dataflow control. It generates, based on the operation mode, the tile width and the output channel number, trigger signals for the address controllers, and the control signal generator. The timer generates five nested trigger signals. W_cnt_trig, kernel_cnt_trig, and line_cnt_trig are built from addressable shift registers with a depth of 16. Figure 5.4 visualizes how the dataflow control uses the trigger signals to build up the data flow for conv2d operations with a kernel size of 3×3, stride 1 and same padding. A, B and P represent the corresponding DSPs registers. The different colors show which input belongs to which output pixel. Float8 requires bringing both values to the same exponent before the summation of two values. For this reason, float8 alternately computes three output pixels. That leads to a more complex data flow for float8 in comparison to int8.

Col_cnt_trig indicates that the tile borders are reached and reinitializes the read addresses. Iter_cnt_trig shows that all iterations of the current tile are done. The number of iterations indicates how often the same input data is sent to the systolic array to compute multiple output channels using the same processing element. For example, if a layer has 64 output channels and tile recurrence is 1, the hardware accelerator requires 4 iterations to compute all output channels. The address controllers increment or decrement their output value based on the trigger signals and the operation mode. The memory checker observers the write address of the processing element cache and the corresponding tail. The tail indicates the first occupied address, the head the last occupied address. If the memory checker detects that the PE cache

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

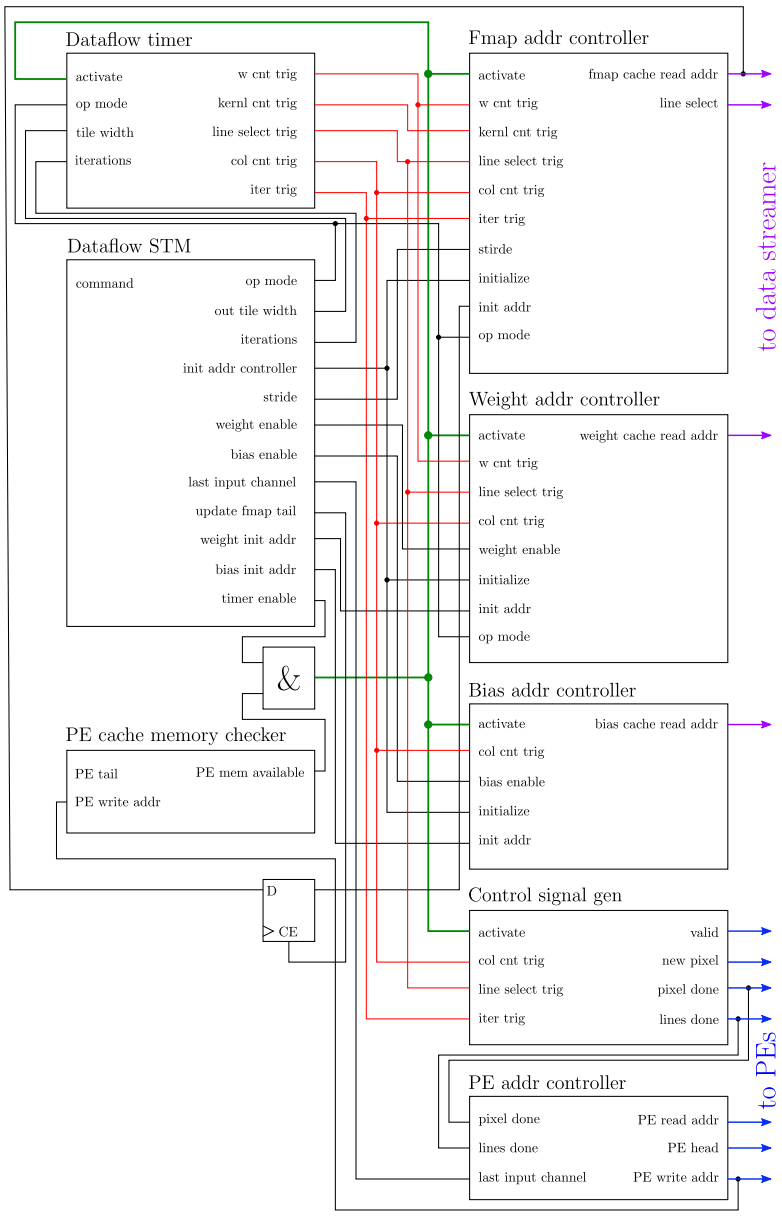


Figure 5.3: Block diagram dataflow control. The diagram shows how the dataflow control generates the read addresses and the control signal to follow a given computation sequence. Dependable of the operation modes, the dataflow timer generates trigger signals which force the address controllers to increment or decrement their output address by a value given by the operation mode

is full, it stops the dataflow timer, the address controllers, and the systolic array by forcing the valid signal to zero.

Figure 5.5 shows the data flow required by a single PU for the computation of a 3×3 convolution with stride 2 and same padding using the float8 implementation. It further visualizes that in comparison to stride 1, the half number of output rows are produced, using an equal number of input rows. However, this would lead to that only half of the PUs would be utilized. For this reason, it is necessary to store twice as many rows using stride 2. Since the PEs use every second row simultaneously, two rows can be stored in one physical input cache line. Therefore, a full PU utilization is still possible using stride 2, since twice the input tile height can be used.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

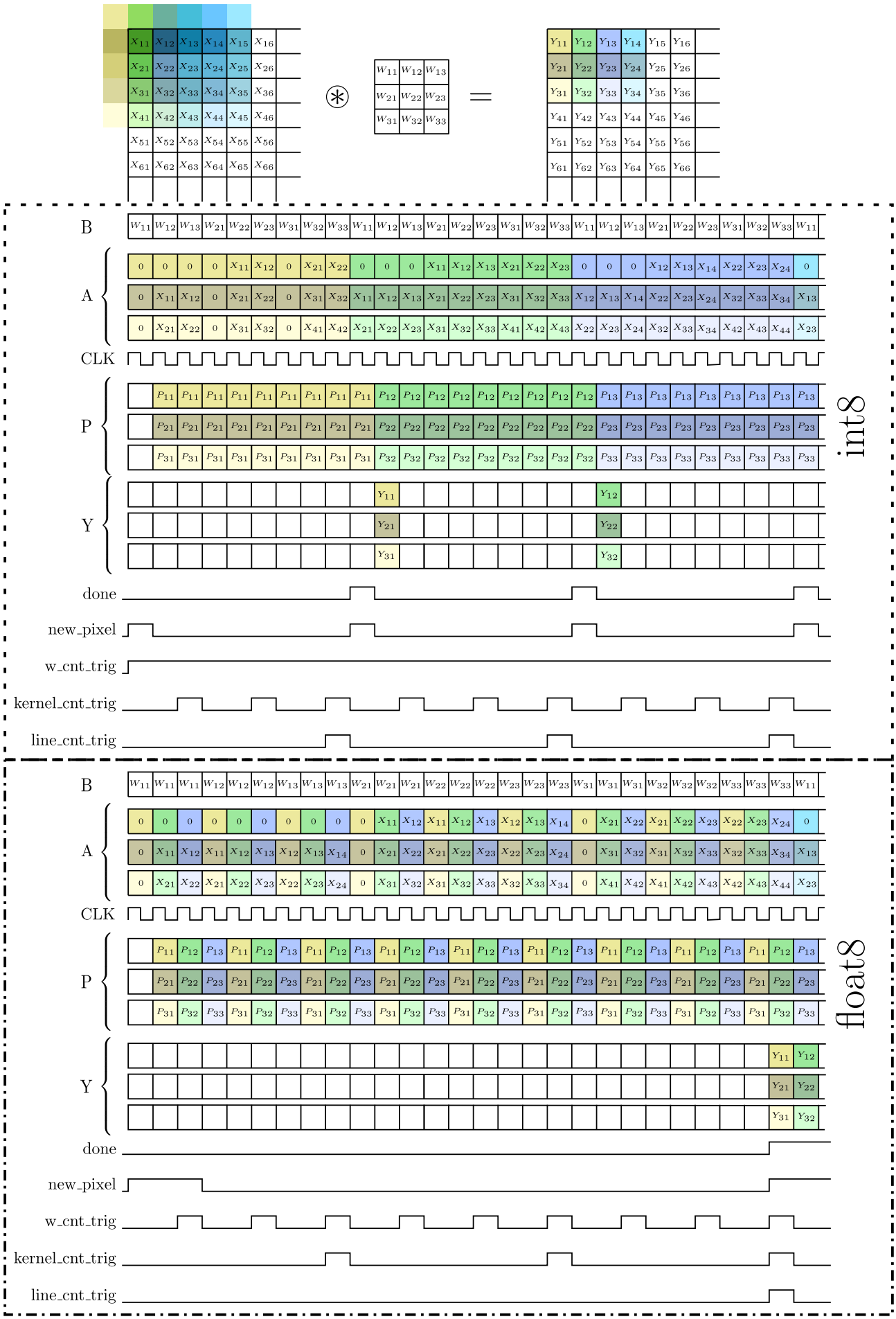


Figure 5.4: Comparison of data flow using int8 (above) and float8 (below) implementation. Float8 computes 3 outputs alternating, while int8 computes one output after each other. The alternating computation of float8 is necessary since the interim results have to be shifted to fit the new exponent. A, B, P represent the corresponding DSP registers. The different colors show which input belongs to which output.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Figure 2 and Figure 3 in Appendix A visualize the data flow used for 5×5 convolution and 5×5 convolution with stride 2, which are similar to the data flow used for 3×3 convolution.

Figure 1 visualizes the data flow for a 1×1 convolution. It shows that the DSP multiplies one input pixel after each other width of the same weight.

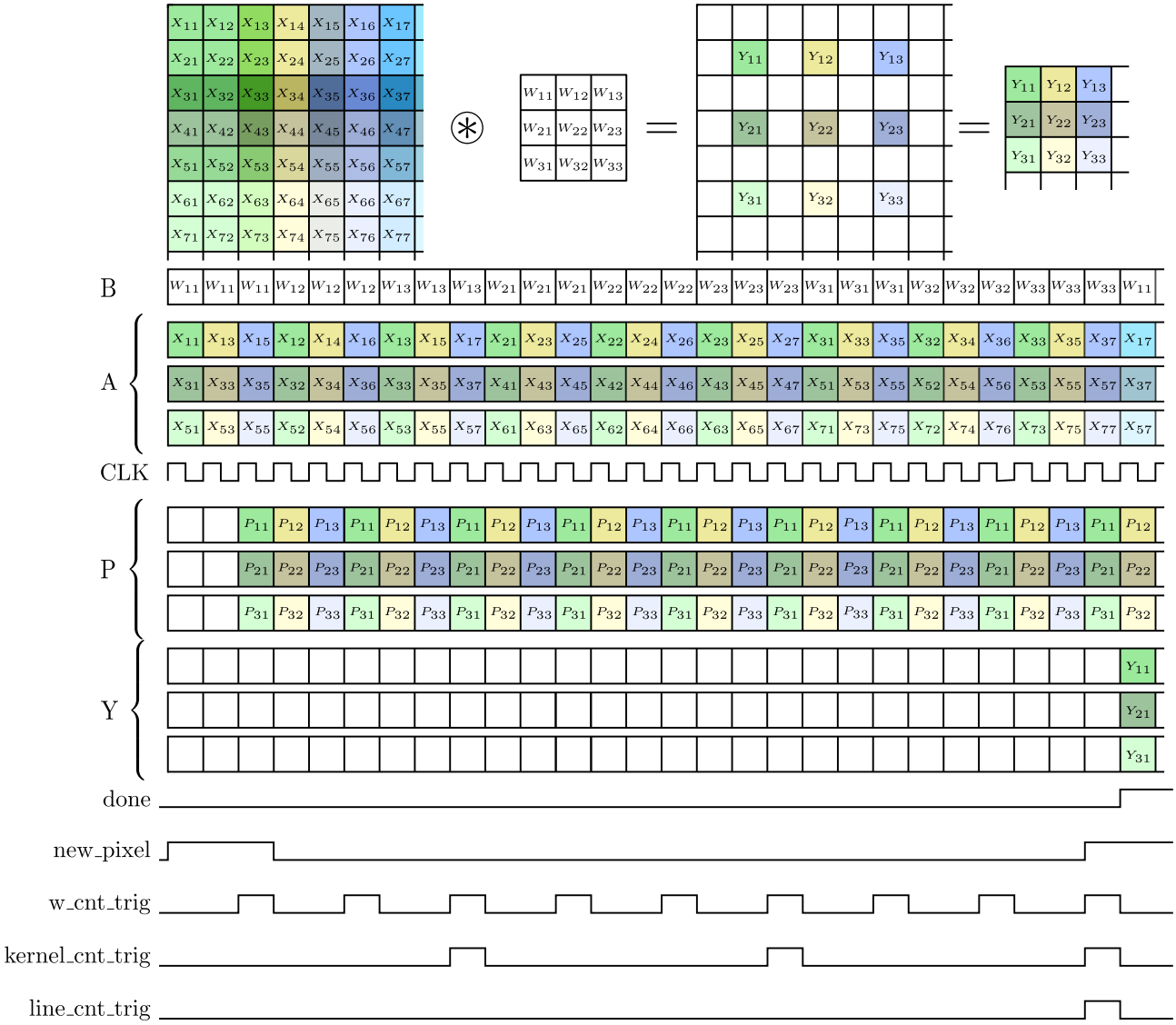


Figure 5.5: Data flow for 3×3 convolutions width stride 2 using the float8 implementation. The colors highlight which input data is used to compute a specific output pixel. Due to the delay of 3 clock cycles caused by the pipelining of the DSP and the floating-point extension, 3 output pixels are computed alternating.

5.2.1 Available and possible future operation modes

The type of convolutional layer and the used kernel size defines the operation mode. Depending on the operation mode, the data streamer unit reads data in a specific sequence from the feature-map cache and the weight BRAMs and passes it to the processing units. The accelerator implements three operation modes required for an object detector. Each of them supports a stride of either (1,1) or (2,2):

- 5×5 convolution
- 3×3 convolution

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

- 1×1 convolution

When designing the hardware accelerator, particular importance is attached to the fact that it is easy to extend the functionality. The following examples intend to show how to extend the functionality:

- inverse Bottleneck 5×5
- inverse Bottleneck 3×3
- fully connected

Inverse bottleneck

Inverse bottleneck layer[3] use a combination of conv2d layer with a kernel size of 1×1 to find the correlation in the depth space, and depthwise separable convolutions for features in x and y direction. The idea is to reduce the amount of required computations by a separation of depth space and xy space. Many existing implementations[4] show that inverse bottleneck layer reduce the number of required operations and the parameter size while preserving or improving the accuracy. However, if one compares the required number of memory accesses, one will notice why object detectors based on inverse bottleneck layers achieve a lower frame rate than comparable object detectors that mainly use conv2d. When using conv2d, the hardware loads the feature-maps from the external memory, executes the computations, and loads the new feature maps back to the external memory. Compared to that, when using an inverse bottleneck layer, the hardware loads the feature map from the external memory, executes conv2d with kernel size 1×1, and loads the expanded feature-maps back to external memory. After that, the hardware loads the feature maps again from the external memory, executes the depthwise separable convolution, and loads the feature map back to the external memory. Next, the hardware performs another conv2d with kernel size 1×1, including all external memory accesses. That leads to more than three times the memory accesses required for loading the feature maps compared to a standard conv2d when using an inverse bottleneck layer.
Therefore, the focus of the implementation of inverse bottleneck layers has to be on reducing the required memory accesses. This can be done as follows:

1. Do a standard conv2d 1×1 for the current tile and keep the results in the PE caches
2. Use the interim results of conv2d 1×1 in the PE caches as input fo the depthwise separable convolution
3. Load the interim results back to the external memory
4. Redo these operations for all tiles
5. Do a standard conv2d 1×1 for all tiles

That workflow reduces the required external memory accesses for loading the feature maps by half. However, to follow this suggested workflow, the PE X-inputs need an additional MUX for selecting either the input of the PE cache or from the data steamer. In addition, the dataflow control has to forward the read address to the PE caches.
The required memory accesses can be reduced even further if the output tile of the depthwise separable convolution is redirected back to the input of the hardware accelerator. However, this would require an additional FIFO and some MUXes to redirect the input and output stream. Nevertheless, that extension is not only useful for the inverse bottleneck layers, but also for all convolution layers followed by a conv2d layer with a kernel size of 1×1.

[3] See Section 2.4.3
[4] See Section 3.6

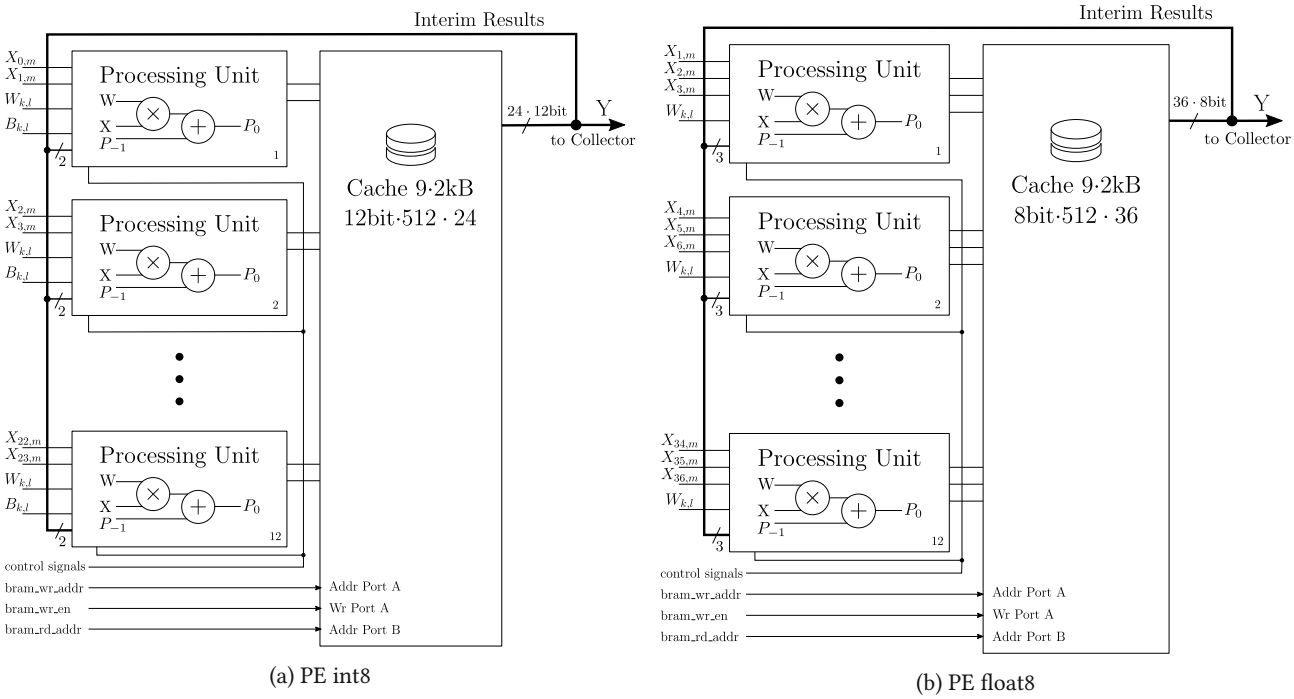
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Fully connected

In contrast to convolutional layers, fully connected layers use different weights for each neuron connection. However, the hardware accelerator does parallelization by reusing the same weight for all PUs in a PE. Using maximum tile recurrence allows using four different weights for each PE. That leads to a maximum of 64 parallel computations for the hardware accelerator when using fully connected layers. The int8 implementation achieves maximum utilization of 16%, whereas float8 achieves 11%. For fully connected layers, there is no parallelization advantage when using float8 in comparison to int8.

However, poor hardware utilization does not necessarily lead to a lower throughput since fully connected layers have a large number of different weights in comparison to convolutional layers. For this reason, the external memory bandwidth is expected to limit the throughput. In addition, the flattened input feature map has to be loaded only once from external memory, which has a positive effect on the required memory bandwidth.

5.3 Processing element



(a) PE int8

(b) PE float8

Figure 5.6: Block diagram processing element int8 implementation (a) and float8 implementation (b).

Figure 5.6 shows a block diagram of the PEs for the int8 (a) and the float8 (b) implementation. In both implementations each PE consists of 12 PUs and 9 18k BRAM slices. The main difference of the implementations is that for float8 each PU can perform 3 MAC operations in parallel, while the int8 implementation can only do 2 MAC operation in parallel. That leads to a maximum tile height of 36 for the float8 implementation and 24 for the int8 implementation. However, for both implementations, the same BRAM number is available, which leads to the fact that for the int8 interim results with a width of 12 bit can be stored, while for float8 only 8 bits are available for the interim results. Based on the data flow generated by the dataflow control, the data streamer forwards the activations, weights, bias, the control signals, and the read- and write addresses for the PE cache to each PE. If the computation of a column is completed, the dataflow control increases the PE cache head address. The PE cache head indicates which data the PE is allowed to transfer to the collector. When the data is transferred to the collector, the PE frees the memory area by increasing its PE tail. The

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

PEs are completely controlled centrally by the data flow control unit. A centralized control has the advantage that the hardware is required only once, which saves resources.

5.4 Processing unit int8

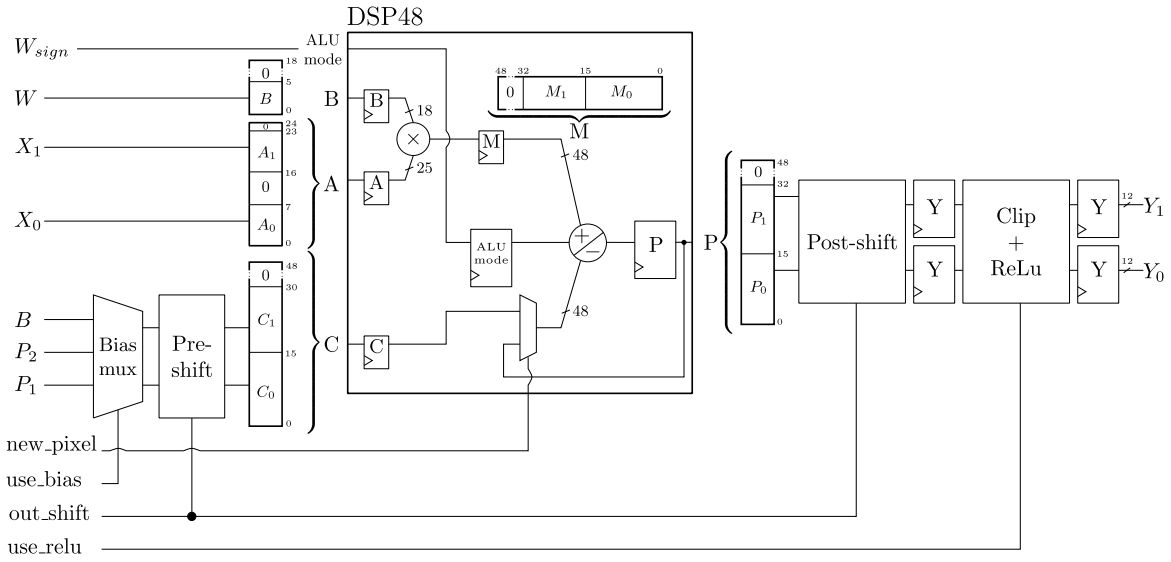


Figure 5.7: Processing unit int8 implementation. It utilizes a DSP48 primitive in combination with LUT-based pre-shift, post-shift, clip and ReLU modules.

Figure 5.7 visualizes the structure of the int8 implementation of the processing unit. The core element is a DSP48 primitive, which implements a 25-bit times 18-bit multiplier and a 48-bit accumulator. Since such a high resolution is not required, the DSP is wired so that it does two 5 times 7-bit unsigned multiplications and two int16 accumulations in parallel. That can be done by dividing the registers A and C into two parts. Using signed activations is not supported since a possible carry of M_0 distorts M_1. However, using ReLU as activation function eliminates that issue since it sets all negative activations to zero. The pre- and post-shift modules support a shift of 0 to 4 bit. The clip module clips the resulting activation to an int12 value. When the use_relu flag is high, ReLU is applied to the output of the processing unit.

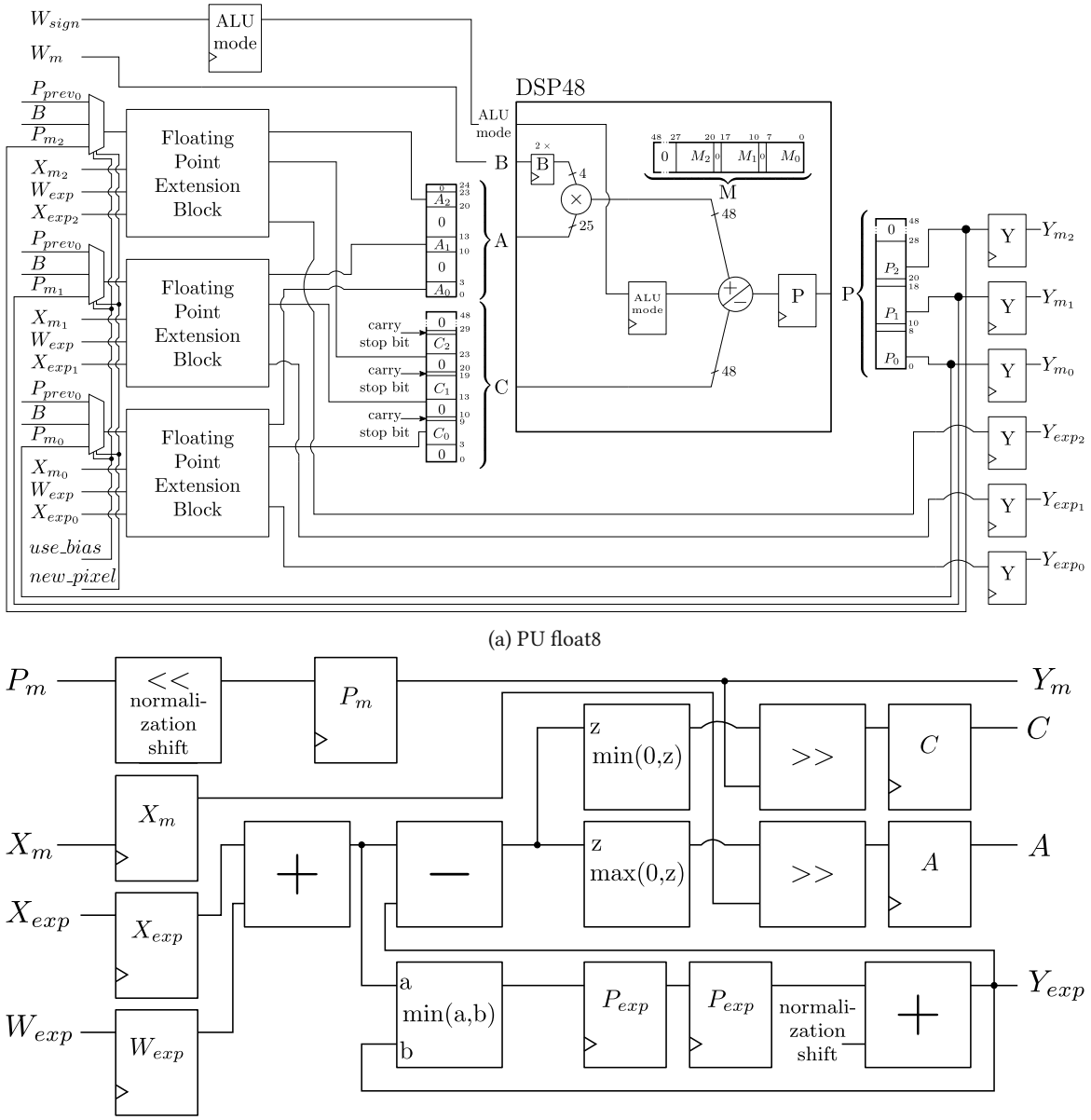
The processing unit can operate with a maximum frequency of 250MHz, compared to related implementations listed in Table 4.3, a frequency of 250MHz is comparatively fast. A drawback of this implementation is that it has to use ReLU as non-linearity after each layer, which is outdated since most modern CNNs use activations such as LeakyReLU or Mish. However, since this disadvantage is accompanied by doubling the parallelization, it is acceptable.

Algorithm 1 PU int8 convolution

```
output = b_q
for c in C_i do
    output « Preshift
    output += conv2d(input[c],w_q[c])
    output » Post-shift
    clip output to 12-bit
end for
output » Postshift
output » 4
clip output to 8-bit
ReLU
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

5.5 Processing unit float8



(a) PU float8

(b) Floating point extension

Figure 5.8: Processing unit float8 with floating point extension. The float8 PU (a) uses a DSP48 primitive in combination with 3 LUT-based floating point extension blocks (b) for float8 MAC operations.

Figure 5.8b visualizes the structure of the float8 processing unit. As with the int8 implementation, the core element is a DSP48 primitive. However, the float8 processing unit uses a reduced multiplication width of 4 bit times 4 bit and a reduced accumulation width of int9. That has the benefit that it can perform three MAC operations in parallel. Separation zeros and a carry stop bits added to the C register avoid the carry of M_0 and M_1 to distort M_1 and M_2. The maximum output bit width of a single multiplication plus a carry stop bit determines the number of separation zeros. Just like the int8 implementation, the float8 implementation supports unsigned input activations only. For this reason, the float8 processing unit uses ReLU as a non-linear activation function.

Due to the small dynamic range of uint4 quantization, an additional three-bit exponent is used to increase the dynamic range of the activations. Additionally, the float9 quantization uses a sign bit to support multiplications with signed weights. The weight quantization uses float6 with one sign bit, one exponent bit and four mantissa bits. Figure 5.8b visualizes the functionality of the floating-point extension block, which does the comparisons and shifts required for

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

floating-point accumulation. The first step is to normalize the previous result for optimal use of the limited mantissa bit width. After that, it subtracts the sum of the weight and the input exponent from the intermediate result exponent to figure out the mantissa shifts. The sign of the subtraction determines which exponent is smaller. Since the exponents are assumed to be negative[5], the smaller exponent is used for the exponent of the new output. The floating-point extension uses two cycles to do the exponent comparison and the shift. For this reason, the float8 implementation requires a specific processing sequence described in Section 5.2, which does an alternating computation of three output pixels.

The floating-point extension is implemented in the fabric part of the FPGA and therefore does not require an additional DSP. However, an efficient implementation is necessary to avoid running out of LUTs. The float8 processing unit achieves a maximum operation frequency of 200MHz, which is slightly slower than the maximum frequency of the int8 implementation. However, it can do 1.5 times more MAC operations each cycle.

Algorithm 2 PU float8 convolution

```
1:  for c_o in C_o do
2:      for f_h in Feature map height do
3:          for f_w in Feature map width do
4:              P_mant = b_q[c_o]
5:              P_exp = b_exp
6:              for c_i in C_i do
7:                  for k_h in Kernel height do
8:                      for k_w in Kenrel width do
9:                          A = input_mant[c_i, f_h, f_w]
10:                         exp_xw = input_exp[c_i, f_h, f_w] + w_exp[c_o, c_i, k_h, k_w]
11:                         exp_diff = P_exp - exp_xw
12:                         if exp_diff > 0 then
13:                             P_mant »= exp_diff
14:                             P_exp = exp_wx
15:                         else
16:                             A_mant »= abs(exp_diff)
17:                         end if
18:                         M = A * w_mant[c_o, c_i, k_h, k_w]
19:                         P_mant += M
20:                         normalize P
21:                     end for
22:                 end for
23:             end for
24:         end for
25:     end for
26: end for
27: ReLU
```

5.6 Data collector

Figure 5.9 shows a block diagram of the data collector. When starting the computation of the last input channel of a tile, the data streamer passes the feature map command to the data collector. The Collector-control unit uses the tile size included in the feature map command to load a tile from the PEs. It loads the output channels of a tile from the processing elements by selecting the PEs in turn till all channels are completely loaded. The data collector stores the fetched data in the Buffer-FIFO, which has an input width equal to 8-bit times the PE height. Each line of the Buffer-FIFO has a separated write enable signal, which depends on the tile height and the availability of valid data. The Collector-control stores the feature map command in the command-FIFO-0 immediately after extracting the required data. The empty

[5] Absolute values have to be smaller than 1

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

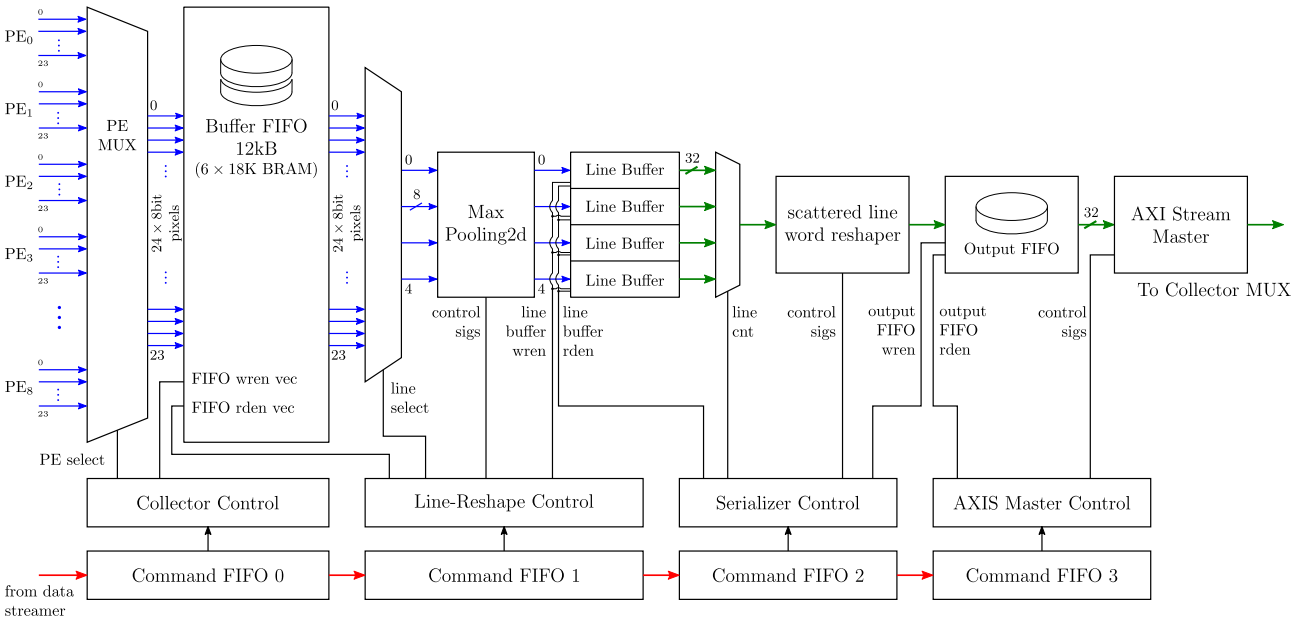


Figure 5.9: Block diagram of the data collector. The data collector fetches the results from the PEs and buffers them in a FIFO, which has an equal number of lines as the PEs. Next, the data is pipelined through the reshaping and serialization stages to transfer the data line by line to the external memory.

signal of the command-FIFO-0 turns low.

The Line-reshape-control waits for the empty signal of command-FIFO-0 to get low and reads the new command from the FIFO. The Line-reshape-control reads four lines simultaneously from the Buffer-FIFO and passes them to the Max-pooling2d unit. It is necessary to read four lines concurrently, as each 32-bit FIFO contains four lines combined in a 32-bit vector. If the max-pooling flag in the command is high, the Max-pooling2d unit performs two maximum pooling 2d operations with strides of 2 every second cycle and stores the remaining two results in two Line-Buffers. When the max-pooling flag is low, the Max-pooling2d unit passes the data directly to the line buffers. The Line-Buffers are modified FIFOs. The difference is that the Line-Buffers combine 4 pixels of a line into a 32-bit vector before storing it in the FIFO. Again, the Line-reshape-control stores the command in a FIFO immediately after extracting the required information.

When a new command is available for the Serializer-control, it reads the output channels of a tile line by line from the Line-Buffers. There are two different serialization scenarios. The first scenario is that the tile width of the current tile is smaller than the feature map width. In that case, the DMA writes the data line by line to the external memory using the scatter-gather feature. That feature allows the DMA to process a list of DMA operations. Each line of the tile corresponds to DMA operation in the scatter list. In that scenario, the data needs to be word-aligned after each line. That means that the last 32-bit vector of a line is filled up with zeros so that the next 32-bit vector starts with the first pixel on position zero.

The second scenario is that the tile width is equal to the feature map tile width. In that case, the lines are transferred continuously to the external memory by the DMA. That has the advantage that a single DMA operation is sufficient to transfer all lines of a tile, which increases the throughput of the DMA. However, in that scenario, the DMA expects the data as one chunk. Therefore, the data must not be word-aligned after each line.

The Scattered-lines-word-reshaper handles the different scenarios. The scattered lines flag included in the command

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

indicates which operation mode the Scattered-lines-word-reshaper uses for the current tile.

The last stage of the data collector is the AXI-Stream master. It forwards the data to the Collector-MUX using the AXI stream protocol. The last signal is high at the last pixel of each output channel, which triggers an interrupt of the AXI-DMA. The interrupt notifies the hardware accelerator driver that the channel is available in the external memory.

5.7 Absolute maximum throughput

The absolute maximum throughput indicates the maximum throughput a hardware accelerator can achieve if all PEs are fully utilized and operate on maximum clock speed. N_{ops} indicates the number of operations each PU can perform in one cycle, which is 4 ($2 \cdot 2$MAC) for the int8 implementation and 6 ($2 \cdot 3$MAC) for the float8 implementation. The processing units achieve a clock frequency of f_{max} of 250MHz using the int8 implementation and a frequency of 200MHz using the float8 implementation. The hardware accelerator utilizes 192 PUs, which are divided into 16 PEs. That leads to the following equation for computing the absolute maximum throughput of the hardware accelerator.

$$OPS_{max} = N_{ops} N_{PU} f_{max} \tag{5.1}$$

When using the int8 implementation:

$$OPS_{max_{int8}} = 4 \cdot 192 \cdot 250\text{MHz} = 192\text{GOPS} \tag{5.2}$$

When using the float8 implementation:

$$OPS_{max_{float8}} = 6 \cdot 192 \cdot 200\text{MHz} = 230.4\text{GOPS} \tag{5.3}$$

5.8 Maximum achievable PE utilization

One problem with the absolute maximum throughput is that it does not take the transfer rate of the external memory and the involved buses into account. The maximum achievable PE utilization targets that issue by computing the input transfers to parallel MAC operation ratio (TOR). A TOR lower than 1 indicates that the number of input transfers exceeds the number of new data required to fully utilize all PEs. On the other hand, a TOR higher than 1 indicates that the transfer rate is sufficient to fully utilize the PEs. Equation (5.4) shows how to compute the number of 32-bit transfers N_{Ti} for a layer with C_i input channels, C_o output channels, a feature map height H and width W, and a kernel size of $K_H \times K_W$. Equation (5.5) computes the required MAC operations N_{MAC} for computing an output feature map of a convolution layer. Based on that, Equation (5.6) computes the number of MAC operations for each PE $N_{PE_{MAC}}$. It shows the equation when using the int8 implementation with 24 parallel lines and 16 parallel PEs. OCR indicates the overclocking of the PEs compared to the bus, which transfers the data to the PEs. Equation (5.8) shows how to compute the TOR for a single layer.

$$N_{Ti} = \frac{C_i \cdot H \cdot W \cdot 8bit + Ci \cdot Co \cdot K_H.K_W \cdot 6bit}{32bit} \tag{5.4}$$

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

$$N_{MAC} = C_i \cdot H \cdot W \cdot Co \cdot K_H.K_W \tag{5.5}$$

$$N_{PE_{MAC}} = \frac{N_{MAC}}{16 \cdot 24} \tag{5.6}$$

$$OCR = \frac{f_{PE}}{f_{Mem}} \tag{5.7}$$

$$TOR = \frac{N_{PE_{MAC}}}{N_{Ti} \cdot OCR} \tag{5.8}$$

Layer	C_o	C_i	H	W	K_H	K_W	N_Ti	N_{MAC}	$N_{PE_{MAC}}$	TOR_{OC1}	TOR_{OC2}
1	16	3	384	384	3	3	110676	63700992	165888	1,49	0,74
2	32	16	192	192	3	3	148326	169869312	442368	2,98	1,49
3	64	32	96	96	3	3	77196	169869312	442368	5,73	2,86
4	128	64	48	48	3	3	50712	169869312	442368	8,72	4,36
5	256	128	24	24	3	3	73776	169869312	442368	5,99	2,99
6	512	256	12	12	3	3	230496	169869312	442368	1,91	0,95
7	1024	512	12	12	3	3	903360	679477248	1769472	1,95	0,97
8	256	1024	12	12	1	1	86064	37748736	98304	1,14	0,57
9	512	256	12	12	3	3	230496	169869312	442368	1,91	0,95
10	255	512	12	12	1	1	42960	18800640	48960	1,13	0,56
11	128	256	12	12	1	1	15384	4718592	12288	0,79	0,39
12	256	384	24	24	3	3	221232	509607936	1327104	5,99	2,99
13	255	256	24	24	1	1	49152	37601280	97920	1,99	0,99

Table 5.1: Comparison required transfer rate and number of operations.

Table 5.1 shows the TOR for all convolution layers of YOLOv3-tiny when using an input size of 384×384. TOR_{OC1} indicates the input transfer to MAC operation ratio when using equal clock speed for the PE clock f_{PE} and the memory clock f_{Mem}. TOR_{OC2} uses two times the clock speed for f_{PE} than for f_{Mem}. That could be, for example, 250MHz and 125MHz. Table 5.1 visualizes that except for layer 11, the transfer rate of the memory bus is sufficient to fully utilize the PEs when using equal clock speeds. When using double overclocking, except for layer 1, all layers with the size of a kernel of 3 have a TOR higher than 90%. The layers with a kernel size of 1 have a low utilization rate. The reason for that is that only C_o MAC operations have to be done for each input pixel. All in all, however, a high PE utilization can be expected even with double overclocking since the computationally intensive layers have a TOR above or close to one.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Chapter 6

Software Toolchain

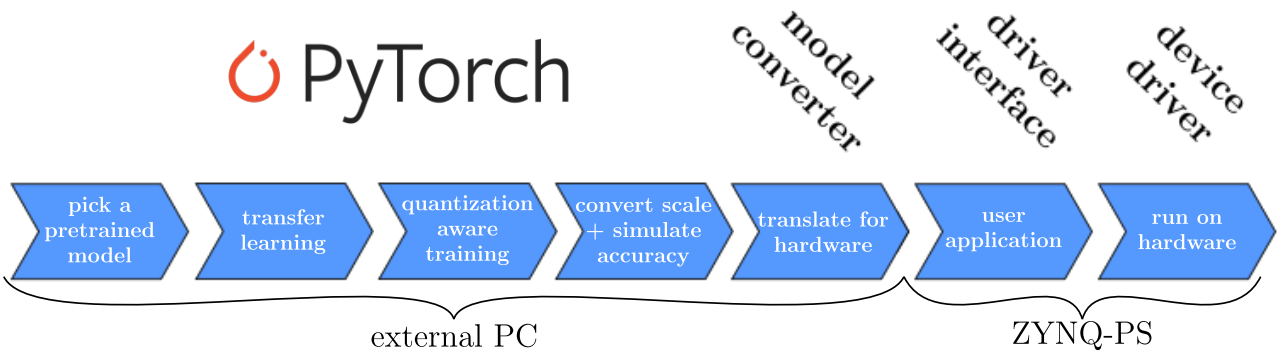


Figure 6.1: Overview of used software toolchain.

Figure 6.1 visualizes the required software toolchain for deploying a neural network on the described hardware accelerator. The starting point for implementing a network in the hardware accelerator is to pick a pre-trained PyTorch model. If there is no pre-trained model for the given task, it is preferable to pick a pre-trained model for a similar task and use transfer learning in the next step. The advantage of transfer learning compared to learning from scratch is that transfer learning reuses the existing features for solving a similar problem. That is done by freezing the feature extraction part of a neural network and retraining the classification part of the model. Therefore, transfer learning reduces the required training time, and a smaller dataset is sufficient. A popular starting point for transfer learning is to use models trained on the ImageNet dataset because the dataset covers many different objects in various scenes. In addition, numerous models are available that are pre-trained on the ImageNet dataset.

The next step is to prepare the network for use in the hardware accelerator. First, all activation functions of non-output layers have to be replaced with ReLU because the hardware accelerator expects unsigned input activations. However, changing the activation functions leads to an accuracy loss. For this reason, retraining is necessary when changing the activation functions. Furthermore, the hardware accelerator does not support batch normalization layers. For this reason, these layers have to be fused into the previous layer. Up to this point, the non-quantized float32 layers are used. Quantization of the weights and biases is necessary to deploy the prepared neural network on the hardware accelerator. The quantized weights, biases, and activation scales are determined using quantization-aware training. That has the advantage that the accuracy loss can be reduced due to retraining of the network considering the quantization error[1]. A

[1] see Section 6.1

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

script transforms the weights and biases into 6-bit integers and the scales into shifts when quantization-aware training is done. Based on the model structure, the input size, and the quantized parameters, the model converter splits the network into processable tiles and generates the commands for the hardware accelerator. The quantized scales, weights, and biases are packed into these commands [2].

The driver interface provides an easy-to-use python interface, similar to PyTorch or Keras, for initializing the device driver using the commands created by the model converter. The device driver is a Linux kernel module that allocates the buffers for each layer and performs the DMA transfers of the commands and tiles. In addition, the kernel driver provides an interface for implementing software layers, such as upsample, max pooling, or residual. This feature increases the flexibility of the hardware accelerator since not every function has to be implemented in the FPGA. Therefore, The hardware accelerator combines the flexibility of the CPU with the parallelization option of the FPGA.

6.1 Quantization aware training

Quantization aware training uses customized PyTorch layers, since PyTorch offers a simple interface to create custom layers and thus enables customizing the quantization to the hardware accelerator. The quantization layer uses an activation scale, a weight scale, and a bias scale to transform the floating-point values into integers. Range trackers determine the scale by searching for the minimum and maximum values of the data. The minimum and maximum value of the weights and biases are determined using a global range tracker. An average range tracker computes the running mean of the maximum and minimum of all input activations using the maximum and minimum in the current batch. The scales are quantized using log2 quantization shown in Equation (6.1). The reason for using log2 quantized scales is that it enables to replace multiplications and divisions with shifts in the FPGA implementation.

$$scale = 2^{\text{round}(\log2(scale))} \tag{6.1}$$

Equations (6.2) to (6.4) show the quantization of the weights $\mathbf{w}$, biases $\mathbf{b}$, and activations $\mathbf{x}$ required to perform a quantized conv2d operation. Using a standard float32 conv2d function in Equation (6.6) is suitable since multiplying and adding integer values result in integer values. The FPGA implementation quantizes the interim results to 12 bits before adding the results of the channels, which is ignored during quantization aware training. The reason for not considering that is that it heavily increases the training time. Equation (6.6) shows the dequantization, which is necessary since the gradient computation uses the default backward conv2d function with $\mathbf{y}$ as input. The orignal weights and biases are not touched by the quantization process since this would eliminate the learning progress.

$$\mathbf{w_q} = \text{clip}(\text{round}(\mathbf{w} \cdot scale_w)) \tag{6.2}$$

$$\mathbf{b_q} = \frac{\text{clip}(\text{round}(\mathbf{b} \cdot scale_b))}{scale_b} \cdot scale_a \cdot scale_w \tag{6.3}$$

$$\mathbf{x_q} = \text{clip}(\text{floor}(\mathbf{x} \cdot scale_w)) \tag{6.4}$$

[2] see Section 6.2

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

$$\mathbf{y_q} = b_q + \text{conv2d}(\mathbf{x_q}, \mathbf{w_q}) \tag{6.5}$$

$$\mathbf{y} = \frac{\mathbf{y_q}}{scale_a \cdot scale_w} \tag{6.6}$$

6.2 Model converter

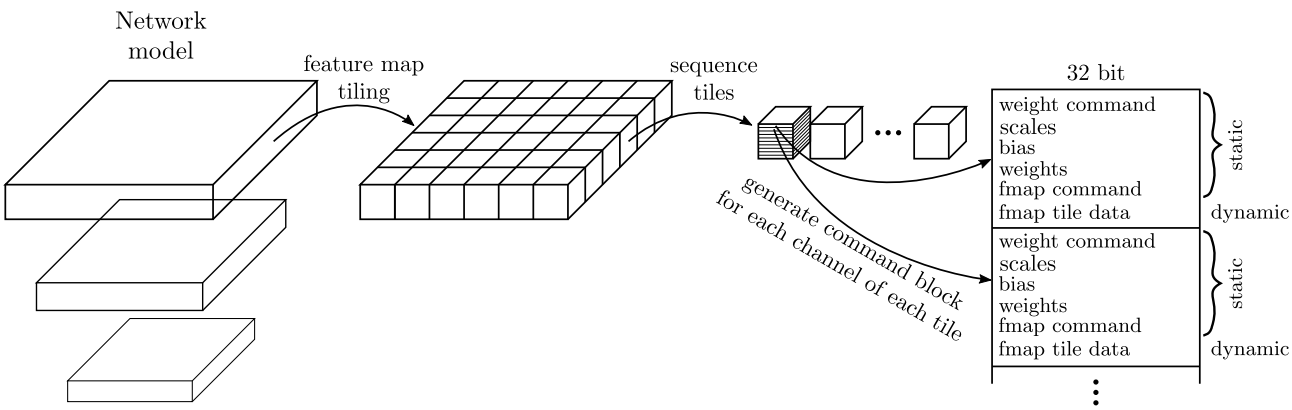


Figure 6.2: Overview of the model compiler workflow.

Figure 6.2 illustrates how the model converter generates command blocks for the execution of a CNN. First it divides the input tensor into processable tiles[3]. For each channel of these tiles the model converter generates a 32 bit weight command and a 32 bit feature map command, which are packed together with the scales, biases, and weights to a command block. Figure 6.2 shows the structure of the static command block.

Table 6.1 shows the structure of the weight command. It includes information about the number of weights, biases and scales required for the computation of the corresponding tile. Besides, it is possible to reuse weights and biases transferred with a previous tile. Therefore, it is necessary to find the desired weight block in the weight cache. For this reason, the destination address of the weight block is included in each weight command. For reusing a tile the weight address is set to the address of the desired block and the new_weight flag is set to zero. Only the command block of the first tile includes the scale since the scale is constant for all tiles within a layer.

Table 6.2 shows the structure of the feature map command and describes the usage. It includes all information to generate the data flow for the current layer. The layer type and kernel size determines the operation mode. Section 5.2.1 describes the currently available operation modes and how additional functionality can be added to the hardware accelerator.

Name	Bit width	Description
weight address	10	Weight cache destination address
reset weight cache	1	Resets tail of weight memory checker
new weights	1	Indicates if the new weights are transferred
weight length	10	Length of the weight block in 32 bit vectors
bias length	7	Length of the bias block in 32 bit vectors
tile recurrence	2	Indicates if the PEs are split into 1,2 or 4 parts
new layer settings	1	Determines whether a new scale is included

Table 6.1: List of all commands included in a 32 bit weight command

[3] See Section 6.2.1

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Name	Bit width	Description
op mode	3	Defines what to do with the received data
stride	1	Stride for convolution operation
tile height	6	Defines the height of a feature map tile
tile width	7	Defines the width of a feature map tile
first channel	1	Indicates that the computation of a new output tile starts
last channel	1	Indicates that the computation of an output tile is done
padding	4	Defines the padding of the current tile.
iterations	5	Defines the number of iterations.
tile recurrence	2	Indicates if the PEs are split into 1,2 or 4 parts.
max pooling	1	Indicates if max pooling is applied to the outputs
scattered lines	1	Indicates if the tile width equals the feature map width.

Table 6.2: List of all commands included in a 32 bit feature-map command.

6.2.1 Feature map tiling

The goal of the feature map tiling is to minimize the required external memory accesses while achieving a high hardware utilization. The hardware accelerator achieves the best PU utilization if the tile height is either the maximum tile height, half the maximum tile height, or a quarter of the maximum tile height[4]. Therefore, the model converter tries using tile heights close to the maximum tile height or its fractions.

For minimizing the external memory accesses, the tile width is chosen so that the intermediate results of an output tile fit into the PE caches. The number of iterations times the tile width determines the size of the intermediate results. The output channel number divided by the number of PEs in the hardware accelerator determines the number of iterations. It indicates how often the hardware accelerator reuses the tile data. For this reason, the ratio of hardware utilization to required external memory accesses for loading the feature map increases with the number of iterations.

6.3 Embedded software

Table 6.3 shows an overview of the software platform. The boot loader[5] and the Linux kernel[6] are modified versions of u-boot-xlnx and linux-xlnx (v4.9) created for Xilinx Petalinux 2017.4. The boot loader is modified to support the used *Zybo-Z7-20* board. In comparison to the Xilinx v4.9 kernel, the Digilent kernel includes an additional DRM encoder to support the Digilent DVI encoder IP, a driver for the OV5640 camera sensor i2c interface, and the MIPI-CSI2 driver. The DRM driver is a custom implementation of Digilent, whereas the camera sensor driver and the MIPI-CSI2 driver are official sources distributed first in the official Linux kernel version 4.14. Instead of the root file system generated by Petalinux, a Linaro distribution based on Debian is used, since it comes with the apt package manager, which facilitates the development process since it supports package downloads form the internet.

	Name	Version
Boot Loader	u-boot-digilent[5]	xilinx-v2017.4
Linux Kernel	linux-digilent[6]	Merge of v4.9 and v4.14
Linux Root Files System	linaro	stretch-developer-20170706-43

Table 6.3: Software platform

[4] see Section 5.1.1
[5] https://github.com/Digilent/u-boot-digilent
[6] https://github.com/Digilent/linux-digilent

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

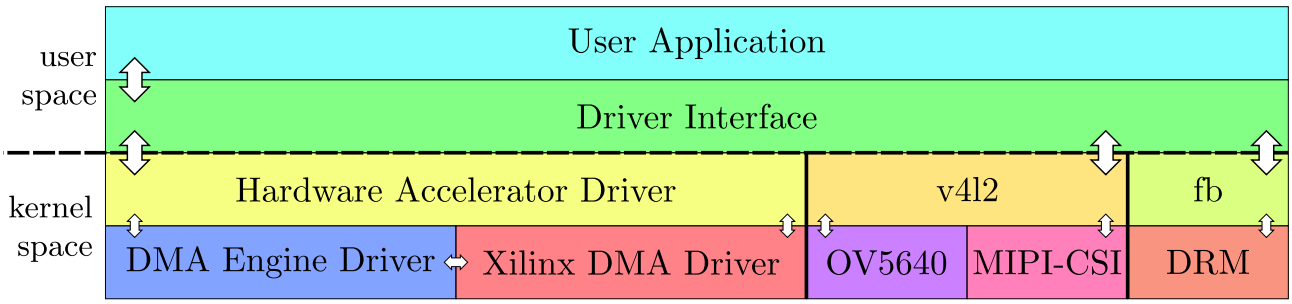


Figure 6.3: Overview of the software abstraction layers.

Figure 6.3 shows an overview of the software abstraction layers. The user application[7] is a python script that configures the network, captures an image, executes the network and plots the results to the framebuffer. This layer is about the structure and the usage of the neural network. The functionality of the hardware is not relevant for the user application. Therefore, the user does not need to know anything about the hardware accelerator design.

The driver interface handles the interaction of the user application and the device drivers. It consists of a C++ core module to interface the kernel drivers and a python wrapper, which offers high-level functions for creating and running CNNs. The wrapper functions are designed to be reminiscent of PyTorch or Keras and can therefore be used natively by neural network designers. In addition, the driver interface provides wrapper functions for capture an image using the video for linux 2 (v4l2) API and for plotting to the frame buffer. The driver interface uses SWIG for interfacing Python and C++.

The hardware accelerator driver, the v4l2 API and the frame buffer API use the ioctl system calls in combination with a memory-mapped buffer for the user to kernel space interfaces. Ioctl system calls provide a request code and optional arguments for controlling and configuring a device driver. The request code is created from an 8-bit driver-specific ioctl number, an 8-bit command specific value, and 13-bit argument length, which leads to a maximum argument size of 8191 bytes. The memory-mapped address space is well suited for image processing applications since it allows to memory map large memory areas between user and kernel space. In addition, it is not necessary to copy the data between user and kernels space, which saves execution time.

6.3.1 Hardware accelerator driver

The hardware accelerator driver is a Linux kernel module that stores the configuration, allocates the buffers, and performs DMA transfers to the FPGA using the DMA-Engine API and the Xilinx DMA driver. A device tree node links the driver to an AXI-DMA IP. When inserting the hardware accelerator driver module, it requests the DMA channels specified in the device tree for RX and TX transfers, allocates and initializes the device property and the interface structure, and creates a character device. The driver interface uses that character device to communicate with the driver from userspace.

The first phase when using the hardware accelerator is to define the structure of the neural network by creating the layers. The driver supports three types of layers: Configuration layers, hardware-accelerated layers, and software-based layers.

Configuration layers provide information for buffer allocation and linking. Examples of configuration layers are input, output, concat, and split. Input layers inform the driver about the size of the input feature map. Output layers notify

[7] See Appendix .2 for an example YOLOv3-tiny user application.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

the driver to link the output buffer with the memory-mapped interface. The concat layer fuses the output buffers of two layers. The split layer splits an output buffer into multiple parts. However, the output channel number of that layer must be a multiple of the part number.

Hardware accelerator layers are executed by the FPGA. Each hardware accelerator layer requires an input buffer, an output buffer, a list of TX tiles, and a list of RX tiles. The input buffer has to be an output buffer of another layer. The input layer creates the input buffer for the first layer, which includes the input size and a link to the memory-mapped interface. The commands created by the model converter include the tiling information. Based on that, the driver generates a list of TX tiles and RX tiles. Each tile has its own scatter list for all necessary DMA transfers. For TX transfers, these lists consist of a pointer to the command block, the command block length, a pointer to the start of each row, and the row lengths. If the tile width is smaller than the maximum tile width, it is possible to transfer the rows continuously. In that case, the DMA requires in addition to the command block entry only one other list entry with the start address and the tile size. Using only a single list entry for the feature map data is beneficial since the execution of each DMA transfer requires an initialization overhead. For this reason, sending or receiving a continuous tiles can be up to two times faster than sending or receiving a tile with scattered lines. An example of a hardware accelerator layer is conv2d. Future hardware accelerator layers can be inverse bottleneck layers or fully connected layers.

Software-based layers are layers that are not executable by the hardware accelerator and are processed by the CPU. The driver provides an interface for implementing software layers in kernel space. The driver executes software layers in a thread, which enables the CPU to start processing a software layer, while the FPGA is still processing tiles of the previous layer. The driver does the synchronization by using completions. Examples of software layers are Maxpooling2d, Upsample, and Copy. A future implementation of residual layers would make the most sense as a software layer since residual layers perform an element-wise summation of two feature maps. For this reason, no reuse of the data is possible or necessary. Therefore, it makes more sense to use the free CPU resources to perform the summations.

When the configuration of the network is done, the driver can execute the network. First, a new input feature map has to be copied to the memory-mapped interface, and ioctl has to be called with the request code for executing the network. Consequently, the driver creates two threads, one for performing the TX DMA transfers and one to perform the RX DMA transfers. These threads start loading the scatter lists into the AXI-DMA using the DMA-Engine API. The Xilinx DMA driver concatenates the scatter lists till the buffer of the AXI-DMA is full or all entries for a layer are loaded into the AXI-DMA. After that, the DMA starts executing the combined scatter list and triggers an interrupt when the execution of all list entries is done. The driver synchronizes the RX and TX threads using completions for each channel. A channel is completed, if all transfers of all corresponding tiles are completed. The tx thread waits with the execution of a channel till the RX thread sets the completion to done. If there are pending transfers and the TX thread is forced to wait, it starts the pending DMA transfers before it starts waiting on the completion.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Chapter 7

Performance Evaluation

The performance evaluation of the hardware accelerator uses the following metrics: accuracy loss, throughput, latency, energy efficiency, and hardware cost. The accuracy indicates the quality of the computed results and is given as the mean average precision mAP of the results[1]. The achievable accuracy depends on the dataset, the object detector, and the quantization method. However, only the quantization method, including the replacement of the activation function, depends on the hardware accelerator presented in this work. Therefore, the accuracy loss due to the used quantization method and the replacement of the activation function is used as a quality measure for the hardware accelerator. The same dataset and the same object detector must be used to ensure a fair comparison of the accuracy loss with other implementations. For this reason, this work uses the COCO dataset and YOLOv3-tiny for evaluating the accuracy loss since this is a common combination.

The latency is the end-to-end processing time for a single detection pass. For better comparison to other implementations, a single inference run of the CNN is used for the latency. Throughput and latency are closely coupled for the presented hardware accelerator since it does not use batching. Therefore, the throughput R can be determined from the latency l and the number of operations for YOLOv3-tiny N_{ops} using the following equation:

$$R = \frac{N_{ops}}{l} \tag{7.1}$$

The energy efficiency indicates the required energy for a single computation and is given in GOPS per Watt or GO/J. In addition, the energy consumption for a single inference run E_{inf} is of interest since it enables an estimation of the energy consumption to accomplish the detection task.

7.1 Quantizing YOLOv3-tiny

Table 7.1 shows the quantizations tasks for creating quantized weights and biases for the presented hardware accelerator from the pre-trained YOLOv3-tiny-weights. The first step for creating quantized weights is to replace the LeakyReLU activation functions with ReLU activation functions. That is necessary since the hardware accelerator requires feature map inputs with an equal sign. However, replacing LeakyReLU with ReLU leads to an accuracy loss. However, using

[1] See Section 3.1

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Task	mAP	mAP diff	Quantization	retrain	epochs	learning rate
0. pick pre-trained model	0.346	-	float32	X		
1. replace LeakyReLU with ReLU	0.326	$\downarrow 2.0\%$	float32	$\surd$	15	$5e^{-5}$
2. fold Batchnormalization layers	0.328	$\uparrow 0.2\%$	float32	$\surd$	5	$5e^{-5}$
3. quantize using val-dataset	0.272	$\downarrow 5.6\%$	int6/int8	$\surd$	5	$5e^{-5}$
4. quantization aware training	0.294	$\uparrow 2.4\%$	int6/int8	$\surd$	10	$5e^{-6}$
5. convert to postscale	0.294	-	int6/int8	X		
6. simulate for FPGA	0.263	$\downarrow 3.1\%$	int6/int8	X		

Table 7.1: Tasks for quantizing YOLOv3-tiny

retraining after replacing the activation functions, the accuracy loss can be reduced to 2 percentage points. The retraining reaches convergence after 15 epochs.

The second step is to fold the batch normalization layers into the previous layers, which does not influence the inference results. However, retraining for 5 epochs is done in this step since the results of the layers have fewer restrictions without batch normalization. The retraining leads to an accuracy gain of 0.2 percentage points.

The next step is to quantize the weights and biases for the validation dataset. In this step, the validation dataset is used for the quantization-aware training. That is done to derive the pre and post-shifts for the quantization-aware training from the validation dataset. The quantization process reduces the mAP by 5.6 percentage points.

Next, some of the lost accuracy is restored using quantization aware training with the training data set. After 10 epochs, the quantization-aware training reaches convergence and restores 2.4 percentage points accuracy. For the quantization-aware training, a lower learning rate of $5e^{-6}$ is necessary since that avoids too many weights from tipping over due to the quantization process.

The quantization-aware training uses pre-and post-shifts and produces floating-point outputs similar to float32 quantization. That is necessary to use the standard float32 backward functions. However, for the FPGA implementation, a conversion back to float32 is not appropriate. Therefore, the next step is to fuse the pre-and post-shifts between two layers to a single post-shift after each convolution. Fusing the pre-and post-shifts does not influence the achievable accuracy.

The last step is to simulate the quantized weights and biases for the FPGA implementation to get the achievable mAP for the hardware implementation. The quantization of the interim results leads to an additional accuracy loss of 3.1 percentage points. That leads to a combined accuracy loss for the int8 implementation of 8.3 percentage points.

The simulation does the identical computations with the same interim quantization bit width as the FPGA implementations. That is verified for the VUNIT simulation and the real FPGA implementation. Therefore, the simulated accuracy corresponds to the accuracy achievable on the FPGA implementation.

The same steps where done using the float8 implementation. However, the aggressive quantization of the interim results leads to a massive accuracy loss. For this reason further investigation are done only for the int8 implementation.

Figure 7.1 shows an example detection using the hardware accelerator and an example image with a size of 383×384. The example image shows seven possible objects, a person, a bench, a dining table, a bottle, a wine glass, a fork, and a knife. The quantized YOLOv3-tiny detects the person, the bottle, the wine glass, and the dining table correctly. However, it does not recognize the fork and the knife. Besides that, it confuses the bench with a chair. However, these are kind of similar objects.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

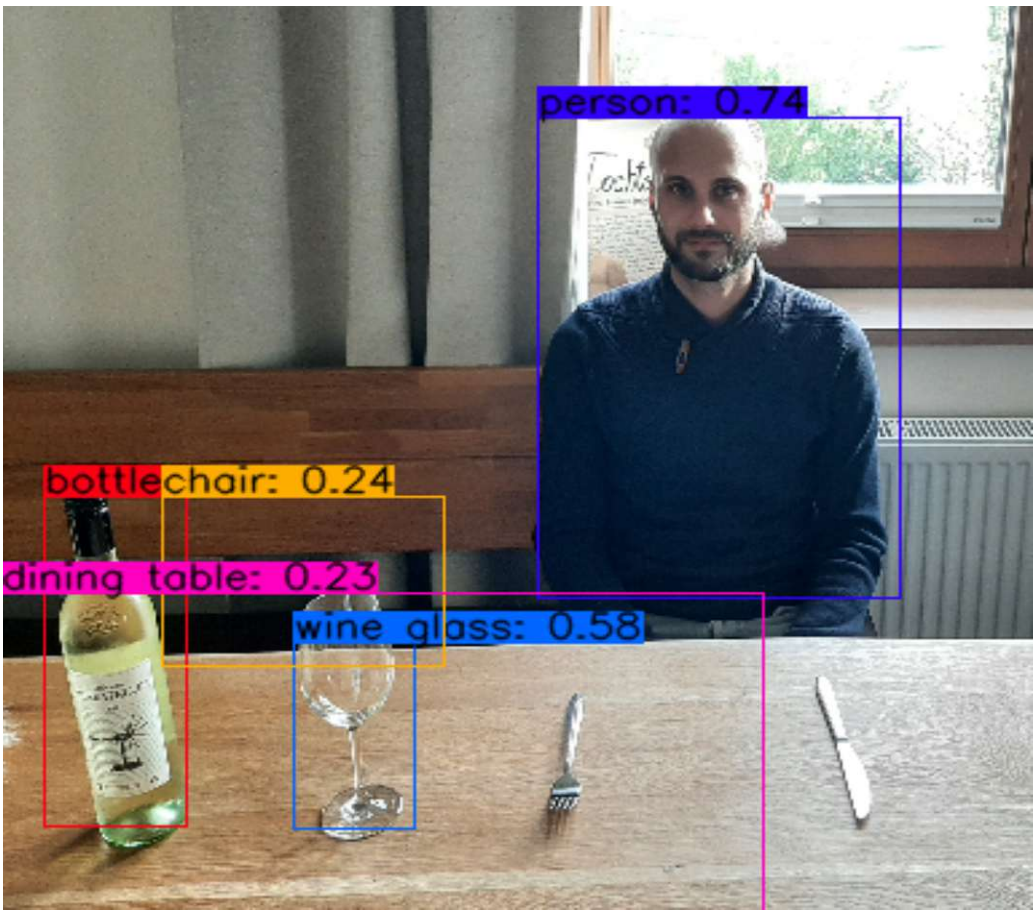


Figure 7.1: Example detection using the hardware accelerator and an example image with a size of 383×384.

7.2 Measuring throughput and latency

Mem Clk [MHz]	PE Clk [MHz]	Latency [ms]	fps	R GOPS	PE Utilization [%]	P_{idle} [W]	P_{exec} [W]	η_{exec} [GO/J]	E_{inf} [J]
100	100	112	8.9	42.41	55.2	2.73	3.12	13.61	0.349
100	175	91	11.0	52.20	38.8	3.32	3.85	13.56	0.351
100	200	89	11.2	53.37	34.7	3.60	4.10	13.03	0.365
125	125	98	10.2	48.53	50.6	3.05	3.42	14.19	0.335
125	200	86	11.6	55.23	35.9	3.57	4.10	13.49	0.353
125	225	84	11.9	56.55	32.7	3.83	4.35	13.01	0.365
125	250	83	12.0	57.23	29.4	4.02	4.57	12.52	0.380

Table 7.2: Latency, throughput, power consumption, energy efficiency and PE utilization using different clock speeds for Mem Clk and PE Clk. It uses YOLOv3-tiny with an image size of 384×384.

Table 7.2 shows the metrics of the hardware accelerator for different combinations of Mem-Clk and PE-Clk. The metrics are determined using YOLOv3-tiny with an image size of 384×384. The reason for using that size is that it leads to feature map sizes which are a multiple of 12×12, which perfectly fits into the implemented hardware accelerator.

For measuring the inference time of YOLOv3-tiny, the high-resolution clock of the std::chrono C++ library is used. The driver interface starts the timer before calling the hardware accelerator driver and stops when receiving the results from the device driver. The average inference time of 1000 runs determines the latency. The number of images processed by the hardware accelerator within one second determines the frame rate. The throughput determines the number of operations the hardware accelerator can process in one second.

The goal of the thesis is to achieve at least 10fps, which is the case for all listed combinations of Mem-Clk and PE-Clk

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

except for using 100MHz for both clocks. Additionally, it shows the resulting utilization of the PEs for each combination used for Mem-Clk and PE-Clk.

Compared to latency and frame rate, the throughput is better suited for a comparison with other hardware accelerators because it is theoretically independent of the used network and input size. In practice, however, each hardware accelerator supports different quantization methods, layer types, and image sizes to a distinct degree, which influence the achievable throughput. For this reason, a fair comparison of the achievable throughput is only possible if both hardware accelerators use the identical quantization and network.

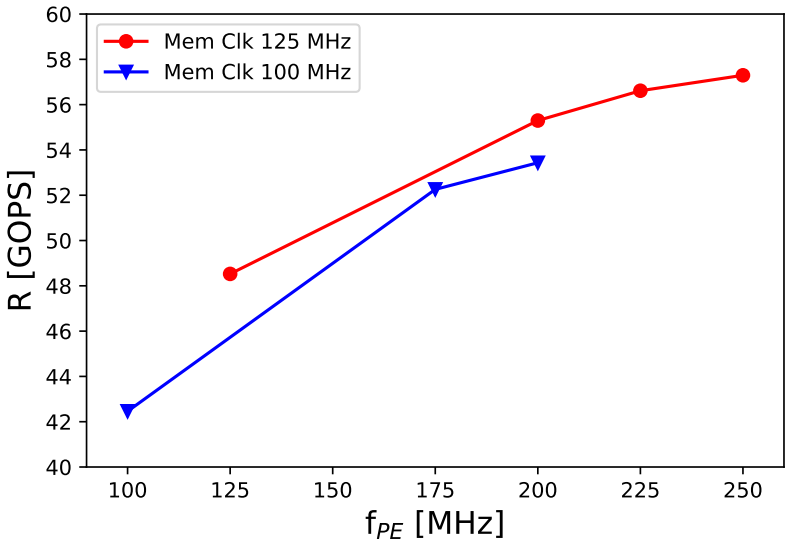


Figure 7.2: Achievable throughput of YOLOv3-tiny dependable of the PE-Clk frequency using a Mem-Clk frequency of 100 MHz (blue) and 125 MHz (red). The throughput is derived from the inference time.

Figure 7.2 visualizes the influence of the clock speeds on the achieved throughput R. The blue curve uses a Mem-clock of 100MHz, the red line uses a Mem-clock of 125MHz. It shows that increasing the clock frequency of the processing elements does not increase the throughput as it would be expected. The reason for that is that the PE utilization decreases with increasing frequency because of the limited external memory bandwidth[2]. It can also be observed that increasing the Mem-clock has a positive effect on throughput. However, again the impact is smaller than expected. As a consequence, it can be observed that a further increase in the PE clock frequency brings less and less throughput gain and approaches a maximum throughput of approximately 58 GOPS. Most hardware accelerators share the problem that the limited memory bandwidth leads to a low processing element utilization for low arithmetic intensity [9]. That behavior is called the roofline model. The shape depends on the ability to reuse the data and the maximum external memory bandwidth. Halawa et. al. show the roofline model of *Nvidia Jetson TK1* and *TX1* for different frequencies [77]. Based on the roofline model, the PE utilization can be derived for the arithmetic intensity of a neural network. Liu et. al. describe the issue of PE underutilization for TPU and Eyersiss and show that TPU achieves a PE utilization of 46.2% for an example CNN used for AlphaGo [78].

Based on the throughput, Equation (7.2) shows the computation of the PE utilization. It is the ratio of achieved throughput R to the absolute maximum throughput $OPS_{max_{int8}}$[3]. Table 7.2 shows the PE utilization for different clock frequencies. The hardware accelerator achieves a PE utilization of 55.2% when using 100MHz for PE-clock and Mem-clock.

[2] For more details see Section 7.3
[3] see Section 5.7

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

With increasing clock frequencies, the PE utilization decreases. When using the maximum clock frequencies of 125MHz for Mem-clock and 250MHz for PE-clock, the hardware accelerator achieves a PE utilization of 29.4%. However, using a faster Mem-clock has a positive impact on PE utilization.

$$\text{PE Utilizaton} = \frac{R}{OPS_{max_{int8}}} \tag{7.2}$$

7.3 Reasons for low PE utilization

Section 5.8 deals with the maximum achievable utilization of the PEs from a theoretical point of view. For that purpose, it uses the relationship between the number of computations a single PE performs for each layer and the transfers required for that.

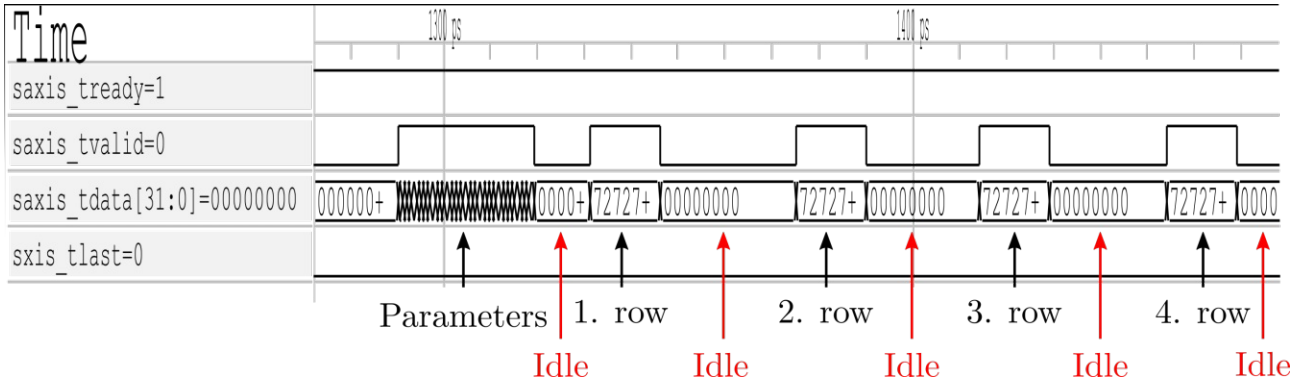


Figure 7.3: Reason for low memory bus utilization for input sizes greater than the maximum tile size.

However, that ratio assumes a perfect utilization of the memory bus. That means that the memory bus transfers data to the PEs without any interruption and never has to wait for new data itself. However, Figure 7.3 shows that this is not the case. It shows the transfer of the parameters and the feature map rows for the computation of the first tile of the first layer. The parameters consist of commands, weights, and biases. They are stored as a contiguous block in the external memory. Therefore, the DMA transfers the parameters as a continuous block without any interruption. However, since the maximum tile width is smaller than the feature map width of the first layer, the rows of the tile are not stored continuously in the external memory. For this reason, transferring each row requires a separate DMA transfer. For each DMA transfer, the DMA uses the address and the transfer length to request a new transfer from the DDR controller. The DDR controller transfers the data from the external memory to the DMA using the AXI-HP bus. Each DMA transfer request consumes time. Additionally, the DMA shares the AXI-HP bus and the DDR controller with other modules. Therefore, the DMA waits until the AXI-HP bus and the DDR controller are available. The idle time depends on the busload. That leads to the idle times marked in Figure 7.3.

There is another reason for long idle times. When the DMA completes all transfers of a scatter-list it sends an interrupt to the CPU and waits till the CPU provides a new scatter-list. These idle times are considerably longer than the idle time shown in Figure 7.3 since the CPU has to stop its current process, save the process data in the stack, and process the interrupt routine before it can transfer the new scatter-list to the DMA.

Therefore feature map tiling with a tile width smaller than the feature map width leads to low utilization of the memory bus. However, for layers with a feature map width smaller than the maximum tile width, a higher utilization of the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

memory bus is achieved. However, the utilization gain is far smaller than expected in Section 5.8. The reason for that are separated DMA accesses for parameters and feature map channels, which interrupts a continuous data transfer. Additionally, initializing transfers for a new layer requires some CPU time since it has to set the completions of the new results, destroy the old transfer threads, check the completions of the data which has to be transferred, and create new transfer threads.

In addition, it is not just a lack of new data that can delay calculations. If the results cannot be transferred back to the external memory fast enough, it leads to a delay in the computations. The reason for that is that the internal memory gets full, which forces the PEs to wait for available memory.

The issues for a low transfer rate from the FPGA back to the external memory are the same as for transferring data from external memory to the FPGA.

7.3.1 Possible improvements of design

Figure 7.3 visualizes that large continuous transfers are crucial to achieving a high transfer rate. A simple way to accomplish that is to increase the maximum tile width. However, increasing the maximum tile width leads inevitably to the problem that the internal PE memories get full more often, and the PEs have to wait till the old results are transferred back to the external memory. To avoid that, the maximum tile width should not exceed half of the PE-cache depth since that would only shift the problem.

Another possibility to increase the transfer rate is to insert a second DMA which handles the transfer of the parameters. First, that would allow transferring larger parameter blocks. Second, it enables a continuous transfer of all channels of feature maps with a tile width smaller than the maximum tile width.

In addition to that, a feedback path from the Data Collector to the Data Streamer could be added. That would allow skipping the transfer to the external memory. However, that is possible only for feature map sizes small or equal to the maximum tile size. When using YOLOv3-tiny with an input size of 384, that would be possible for layers 5, 6, 8, 9, 11, and 12.

Another possibility would be to use the high-performance pins of the FPGA to connect additional external memory. That could solve the issue of limited external memory bandwidth. However, adding additional external memory requires design effort, increases the power consumption and increases hardware costs.

7.4 Energy efficiency

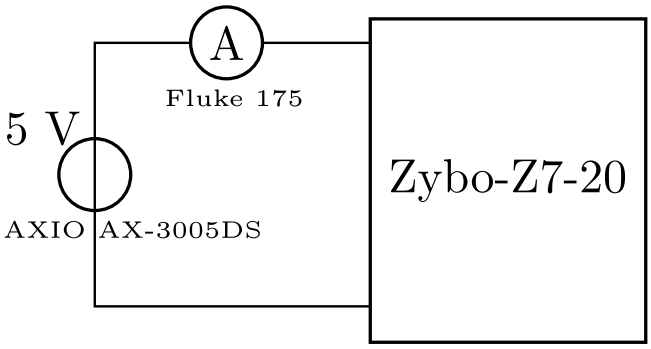
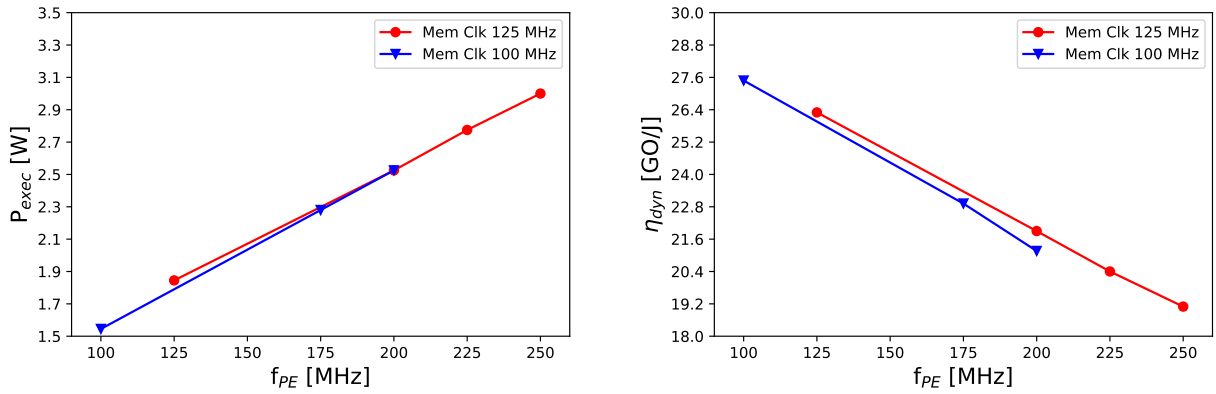


Figure 7.4: Setup for measuring the power consumption of the *Zybo-Z7-20* board.

Figure 7.4 visualizes the setup for measuring the power consumption of the *Zybo-Z7-20* board. However, to determine the energy efficiency of the hardware accelerator, it would be necessary to know the power consumption of the FPGA

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN



(a) (b)

Figure 7.5: Dynamic power consumption (a) and energy efficiency (b) using different frequencies for Mem-Clk and PE-clk.

in combination with the CPU and the external DRAM without all other components on the board. However, such a measurement would require measurement shunts on the board which are not available. Related works solve that issue by subtracting the static power consumption from the power consumption of the board to get the dynamic power consumption P_{dyn} and use that value for computing the energy efficiency [8,70]. The static power consumption P_{static} defines the power consumption when neither the FPGA nor the CPU is programmed. The idle power consumption P_{idle} is the power consumption when both the FPGA and CPU are programmed, but the application is not running.

Equation (7.4) shows the computation of the energy efficiency of the whole board by dividing the throughput R by the power consumption of the board when running the application P_{exec}. In contrast to that, Equation (7.5) shows the computation of the energy efficiency using the dynamic power consumption only. Equation (7.3) shows the measured static power consumption of the board. Equation (7.6) shows the computation of the energy consumption of a single inference run E_{inf} from the power consumption P_{exec} and the inference time t_{inf}. It uses P_{exec} since it includes all required energy for processing the inference of a single detection.

$$P_{static} = 1.575\text{W} \tag{7.3}$$

$$\eta_{exec} = \frac{R}{P_{exec}} \tag{7.4}$$

$$\eta_{dyn} = \frac{R}{P_{dyn}} = \frac{R}{P_{exec} - P_{static}} \tag{7.5}$$

$$E_{inf} = \frac{P_{exec}}{t_{inf}} \tag{7.6}$$

Table 7.2 shows the results of Equation (7.4) and Equation (7.6) for different combinations of Mem-clock and PE-clock. It shows that the energy efficiency decreases when increasing the PE-clock frequency. However, increasing the Mem-clock frequency increases the energy efficiency of the hardware accelerator.

Figure 7.5a visualizes the dynamic power consumption of the hardware accelerator for different PE-clock and Mem-clock frequencies. The blue line uses a Mem-clock of 100MHz, the red line uses a Mem-clock of 125MHz. It shows that the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

power consumption increases linear when increasing the PE-clock. The Mem-clock has a negligible effect on the power consumption. The reason for that is that the main part of the dynamic power consumption is caused by the DSPs, which use the PE-clock. Figure 7.5b visualizes the energy efficiency of the hardware accelerator. Again the blue line uses a Mem-clock of 100MHz and the red line uses a Mem-clock of 125MHz. It shows that the energy efficiency decreases with increasing frequency. The reason for that is the decreasing PE utilization rate for increasing clock frequencies.

7.5 FPGA resource utilization

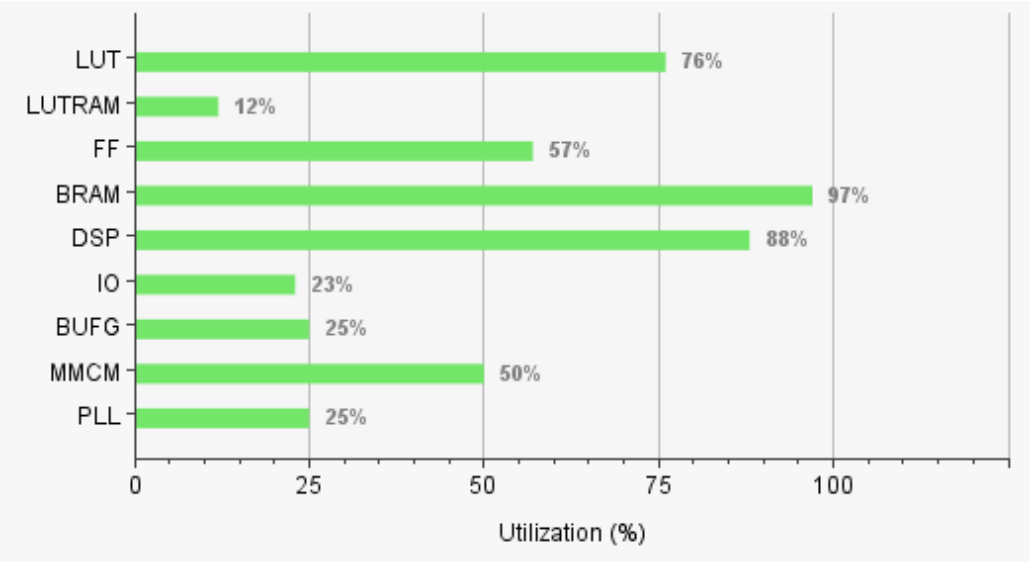


Figure 7.6: FPGA resource utilization of xc7z020clg400-1 using the int8 implementation

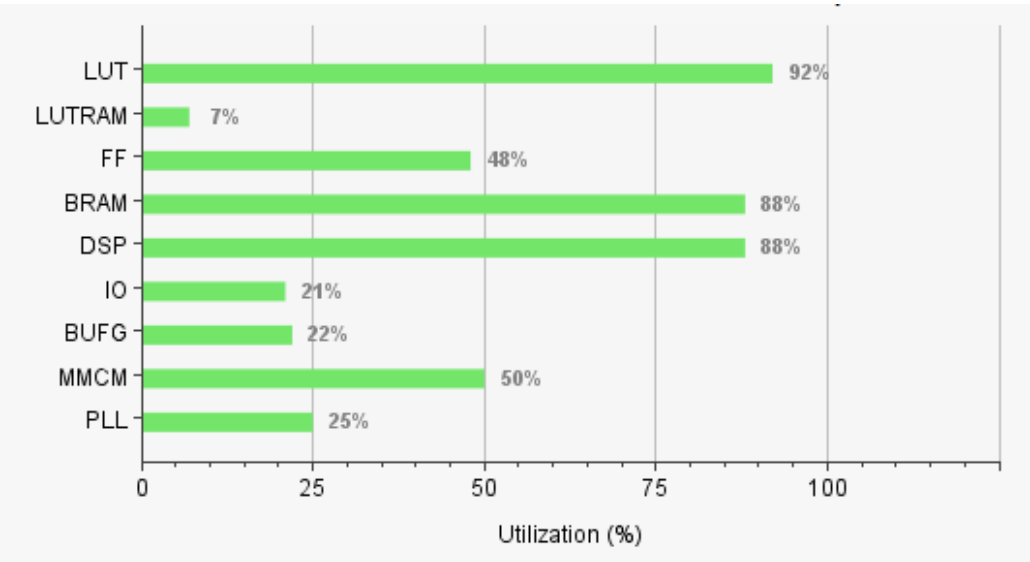


Figure 7.7: FPGA resource utilization of xc7z020clg400-1 using the float8 implementation

Figure 7.6 shows the FPGA resource utilization of the used FPGA (xc7z020clg400-1) using the int8 implementation. The implementation includes the presented hardware accelerator, a MIPI-CSI2 receiver module, and an HDMI encoder. The corresponding block design can be found in Appendix A. The block design bases on the public available *Zybo Z7-20 Base Linux Design* published by Digilent.

Figure 7.7 shows the resource utilization for the float8 implementation using the same block design. Comparing Figure 7.6 and Figure 7.7 shows that both implementations utilize only 88% of the available DSPs. For the float8 imple-

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

mentation, the limiting resource elements are the available LUTs. For the int8 implementation adding DSPs to each PE would be possible. However, that would destroy the symmetry of the hardware accelerator. Moreover, it is not to be expected that the throughput will increase if 13 instead of 12 DSPs are used per PE since the utilization of the PEs is low anyway.

7.6 Comparison to related implementations

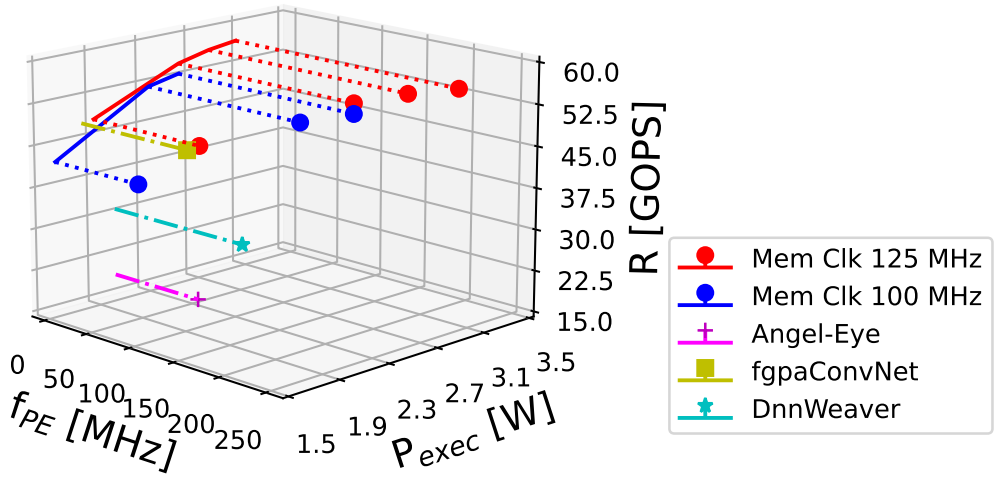
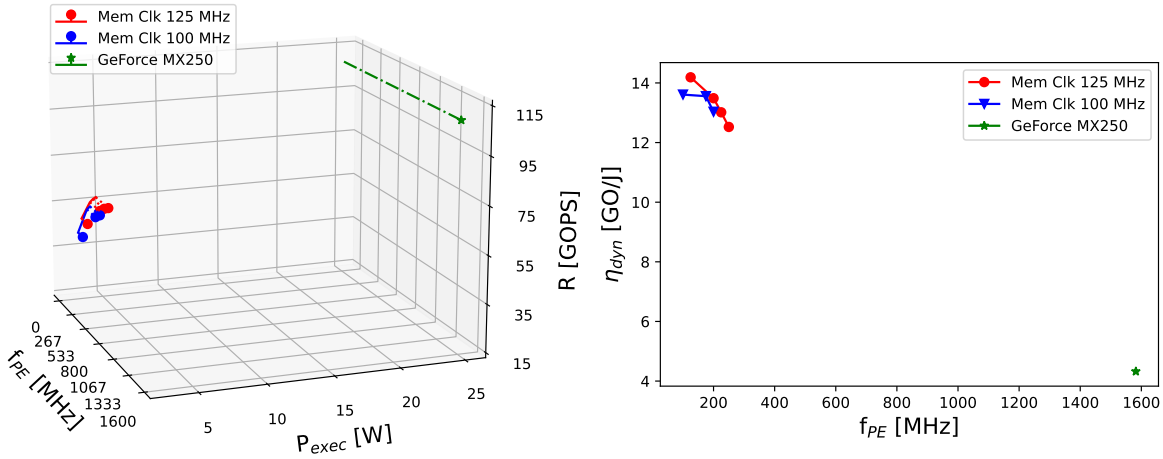


Figure 7.8: Comparison with related implementations in performance space: Frequency, throughput, and power. Data from [8, 70, 79] and Table 7.2

Figure 7.8 shows a comparison of related implementation to the hardware accelerator presented in this work. The blue line and the red line show the performance of the presented hardware accelerator with a Mem-clock of 100MHz (blue) and 125MHz (red). The plus in magenta shows the implementation of Guo et. al., which is called Angle-Eye. It achieves a throughput of 19.2GOPS using a clock frequency of 100MHz and has a power consumption of 2W [8]. The star in cyan shows the implementation of Sharma et. al., which is called DnnWeaver. It achieves a throughput of 31.35GOPS using a clock frequency of 150MHz and has a power consumption of 2W [79]. The square in yellow shows the implementation of Venieris et. al., which is called fpgaConvNet. It achieves a throughput of 48.3GOPS using a clock frequency of 125MHz and has a power consumption of 1.75W [70].

Comparing Angel-Eye with the presented hardware accelerator using equal clock frequencies shows that the presented hardware accelerator achieves a 2.21 times higher throughput while using 77.7% of the power. In contrast to DnnWeaver, the presented hardware accelerator achieves 1.55 times higher throughput and dissipates 92.25% of the power while using only 125MHz instead of 150MHz. Using equal clock frequencies fgpaConvNet and the presented implementation achieve almost identical throughput. However, fpgaConvNet has 5% less power consumption and uses a 16-bit quantization instead of the 8-bit quantization used by the presented hardware accelerator, which allows achieving a lower accuracy loss.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub



(a) (b)

Figure 7.9: Performance (a) and energy efficiency (b) comparison of the presented hardware accelerator and an *Nvidia GeForce MX250* GPU.

7.7 Comparision to Nvidia GeForce MX250 GPU

The *Nvidia GeForce MX250* GPU is designed for use in notebooks and, therefore, for low power consumption. It uses the Pascal architecture with 384 Cuda cores running with a frequency up to 1582MHz and has 2GB external memory connected with GDDR5, which achieves a memory bandwidth of 48.06 GB/s. Using full speed it has a power consumption of 25W. The GPU is assembled in a *DELL Inspiron 13 5391* with an *Intel i7-10510U* CPU.

For a fair comparison, the throughput of the *MX250* GPU is determined in the same way as it is determined for the FPGA. First, the inference time is measured using the default_timer from the timeit python package. The timer starts before calling the PyTorch model and stops when receiving the results. The model uses a 32-bit floating-point quantization, batch size 1, no batch-normalization layers, and ReLU as the activation function. As with the FPGA implementation, the final YOLO layers and the non-max-suppression are not included in the inference time. Based on that, the latency is determined by the average inference time of 1000 runs.

The GPU achieves a latency of 44ms. However, the GPU performance strongly depends on the GPU temperature since the laptop has no sufficient cooling. The GPU reduces the frequency with increasing temperature. Repeating that measurement with an already "hot" GPU leads to an average latency of 68ms.

Figure 7.9a shows a comparison of the measured FPGA and GPU metrics in performance space. It visualizes that the GPU achieves almost double throughput when using batch size 1. However, it has a five times higher power consumption and uses a more than six times higher clock frequency. The comparison uses P_{exec} instead of P_{dyn} for the FPGA since the power consumption of the GPU includes the static and the dynamic power consumption too. However, it does not include the power consumption of the connected CPU which is the case for the *Zybo-Z7-20* board. Figure 7.9 shows the energy efficiency of the hardware accelerator compared to the energy efficiency of the *GeForce MX250* GPU. It shows that the hardware accelerator achieves a 3.28 times higher energy efficiency.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Chapter 8

Conclusion

Object detection based on deep learning has experienced its breakthrough in the last few years. Especially in complex environments, CNN-based object detectors achieve significantly higher accuracy than conventional algorithms. However, a drawback of CNN object detection is that it requires a high computational effort. That impedes its use in edge devices since their CPUs cannot provide the computational power for real-time processing. For this reason, edge devices use hardware accelerators to achieve the required performance. There are three types of hardware accelerators for object detection: GPUs, FPGAs and ASICs. GPUs have the benefit that they are easy to use, provide high throughput, and enable training and inference. The drawback of GPUs is that they have high energy consumption. Compared to GPUs, FPGAs and ASICs achieve a higher energy-efficiency. Besides, FPGAs and ASICs enable the implementation of various interface types and fieldbuses for communication with other machine parts. Compared to ASICs, FPGAs provide higher flexibility since FPGAs consist of programmable hardware. However, a disadvantage of FPGAs is that they achieve a lower throughput compared to ASICs and GPUs. Since FPGAs provide high flexibility in combination with low power consumption and achieve sufficient throughput for real-time object detection, FPGAs are a promising candidate for implementing a real-time capable object detector for edge devices.

For this reason, this thesis presents a hardware accelerator implemented in a low-cost *Zybo-Z7-20* FPGA-board, which achieves a frame rate greater than 10fps using YOLOv3-tiny. The hardware accelerator is embedded into a Linux system, which is running on the processing system part of the Zynq device, by using a device driver. A python wrapper interfaces the device driver and provides high-level functions comparable to PyTorch or Keras. Additionally, this thesis introduces a converter, which enables a translation of a pre-trained PyTorch or Keras model into commands that can be executed by the presented hardware accelerator. Therefore, the presented hardware accelerator provides comparable usability to GPUs.
The hardware accelerator uses an int8 quantization for the activations and an int6 quantization for the weights and biases. It consists of 16 PEs, whereas each PE can perform 24 MAC operations in parallel. The PEs can operate with a clock frequency up to 250MHz. The central data flow control unit controls the data flow for all PEs. It supports conv2d layers with a kernel size of 1, 3 and 5, strides of 1 and 2, and in place max-pooling 2d. The CPU acts as a supervisory controller for the hardware accelerator. It transfers the commands, the network parameters, and the feature map data from the external memory to the FPGA using a DMA.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

The converter uses quantization-aware training for quantizing the weights and biases. The quantization leads to a decrease in the mean average precision of 8.6%. In addition to the int8 PU, this thesis introduces a float8 PU which enables 1.5 times higher parallelization. However, using float8 quantization leads to a high accuracy loss. Therefore, further investigations are pointless.

The hardware accelerator provides adjustable asynchronous clock frequencies for memory and PE-clock. That enables to find an optimal ratio of energy consumption and throughput, depending on the application. Using 100MHz for memory-clock and PE-clock, the *Zybo-Z7-20* board consumes 3.12W and achieves a frame rate of 8.9fps. That corresponds to a throughput of 42.41GOPS. Using 125MHz for the memory-clock and 250MHz for the PE-clock, the hardware accelerator consumes 4.57W and achieves a frame rate of 12fps. That corresponds to a throughput of 57.23GOPS.

In comparison to a power-optimized laptop GeForce MX250 GPU, the hardware accelerator achieves half the frame rate while consuming a fifth of the power. That leads to a 3.28 times higher energy efficiency for the FPGA compared to the GPU.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Chapter 9

Outlook

The presented hardware accelerator offers good usability, can be used flexibly, and achieves sufficient throughput. However, it has comparably high accuracy loss and a low PE utilization. Therefore, the design has a high potential for improvements.

The high accuracy loss has two main reasons. First, the requirement of unsigned input activations requires a replacement of LekayReLU with ReLU. That leads to an accuracy loss of 2 percentage points, which could be avoided. However, a modification of the PU is necessary to support negative and positive input activations. The current implementation does not take advantage of the additional pre-adder provided by the DSPs. That pre-adder can be used to replace the concatenation of X_0 and X_1 with a summation of X_0 with X_1 shifted by 16. However, for each negative X_0, M_1 is decremented, which could distort the results of P_1. However, that could be fixed using post-processing of the results at P_1 [6].
Second, the quantization to 12-bit after the computation of each input channel leads to an accuracy loss of 3.1 percentage points. The issue cannot be solved using the current hardware accelerator design and data flow since there are not enough BRAMs available to increase the bit width of the PE caches. If the feature map tensor would be rearranged to channels first, it would be possible to keep the interim results in the DSP registers till the computation of an output channel is done. However, that would require a redesign of the hardware accelerator

Section 7.3 describes in detail the reasons for the low PE utilization and shows which improvements are possible. Briefly summarized, the main reason for the low PE utilization is that the hardware accelerator requires the transfer of many small data chunks (e.g. rows or channels) from the external memory to the FPGA and vis versa. That leads to idle times between the data chunks, which decreases external memory bandwidth. Increasing the data chunk size reduces the number of idle times. However, since the local cache size is limited by the BRAM number, it is not possible to increase the data chunk size when using large feature map sizes. Again, rearranging the feature map tensor to channels-first could be beneficial since it would lead to larger chunk sizes.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Bibliography

[1] Q. Zhang, L. T. Yang, Z. Chen, and P. Li, "A survey on deep learning for big data," *Information Fusion*, vol. 42, pp. 146–157, 2018.

[2] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[3] X. Xu, Y. Ding, S. X. Hu, M. Niemier, J. Cong, Y. Hu, and Y. Shi, "Scaling for edge inference of deep neural networks," *Nature Electronics*, vol. 1, no. 4, pp. 216–222, 2018.

[4] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "Yolov4: Optimal speed and accuracy of object detection," *arXiv preprint arXiv:2004.10934*, 2020.

[5] G. Mahalingam, K. M. Gay, and K. Ricanek, "Pcb-metal: A pcb image dataset for advanced computer vision machine learning component analysis," *2019 16th International Conference on Machine Vision Applications (MVA)*, pp. 1–5, 2019.

[6] Xilinx, Inc., "Deep learning with int8 optimization on xilinx devices white paper (wp485)," 2017. [Online]. Available: https://www.xilinx.com/support/documentation/white_papers/wp486-deep-learning-int8.pdf

[7] E. Chung, M. Haselman, M. Abeydeera, L. Adams, H. Angepat, C. Boehn, D. Chiou, O. Firestein, A. Forin, K. S. Gatlin, M. Ghandi, J. Fowers, S. Heil, K. Holohan, A. El Husseini, T. Juhasz, K. Kagi, R. K. Kovvuri, S. Lanka, F. van Megen, D. Mukhortov, P. Patel, K. Ovtcharov, B. Perez, A. Rapsang, S. Reinhardt, B. Rouhani, A. Sapek, R. Seera, S. Shekar, B. Sridharan, G. Weisz, L. Woods, M. Papamichael, P. Yi Xiao, D. Zhang, R. Zhao, D. Burger, A. Caulfield, T. Massengill, M. Liu, D. Lo, and S. Alkalay, "Serving DNNs in real time at datacenter scale with project brainwave," *IEEE Micro*, vol. 38, no. 2, pp. 8–20, 2018.

[8] K. Guo, L. Sui, J. Qiu, S. Yao, S. Han, Y. Wang, and H. Yang, "Angel-Eye: A complete design flow for mapping cnn onto customized hardware," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 1, pp. 35–47, 2017.

[9] E. Wang, J. J. Davis, R. Zhao, H.-C. Ng, X. Niu, W. Luk, P. Y. K. Cheung, and G. A. Constantinides, "Deep neural network approximation for custom hardware," *ACM Computing Surveys*, vol. 52, no. 2, pp. 1–39, 2019.

[10] L. Baischer, M. Wess, and N. TaheriNejad, "Learning on hardware: A tutorial on neural network accelerators and co-processors," *arXiv preprint arXiv:2104.09252*, 2021.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

[11] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[12] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *arXiv preprint arXiv:1804.02767*, 2018.

[13] B. L. Deng, G. Li, S. Han, L. Shi, and Y. Xie, "Model compression and hardware acceleration for neural networks: A comprehensive survey," *Proceedings of the IEEE*, vol. 108, no. 4, pp. 485–532, 2020.

[14] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770–778, 2016.

[15] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," *Thirty-first AAAI conference on artificial intelligence*, 2017.

[16] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," *Proceedings of the 36th International Conference on Machine Learning*, vol. 97, pp. 6105–6114, 2019.

[17] K. Abdelouahab, M. Pelcat, J. Serot, and F. Berry, "Accelerating CNN inference on FPGAs: A survey," *arXiv preprint arXiv:1806.01683*, 2018.

[18] A. Shawahna, S. M. Sait, and A. El-Maleh, "FPGA-based accelerators of deep learning networks for learning and classification: A review," *IEEE Access*, vol. 7, pp. 7823–7859, 2018.

[19] J. Qiu, S. Song, Y. Wang, H. Yang, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, and N. Xu, "Going deeper with embedded FPGA platform for convolutional neural network," *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 26–35, 2016.

[20] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *4th International Conference on Learning Representations, ICLR*, 2016.

[21] J. Cheng, P.-s. Wang, G. Li, Q.-h. Hu, and H.-q. Lu, "Recent advances in efficient computation of deep convolutional neural networks," *Frontiers of Information Technology & Electronic Engineering*, vol. 19, no. 1, pp. 64–77, 2018.

[22] Y. Guo, A. Yao, and Y. Chen, "Dynamic network surgery for efficient DNNs," *Advances in neural information processing systems*, pp. 1379–1387, 2016.

[23] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," *Advances in neural information processing systems*, pp. 1135–1143, 2015.

[24] V. Lebedev and V. Lempitsky, "Fast convnets using group-wise brain damage," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2554–2564, 2016.

[25] X. Zhang, J. Zou, K. He, and J. Sun, "Accelerating very deep convolutional networks for classification and detection," *IEEE transactions on pattern analysis and machine intelligence*, vol. 38, no. 10, pp. 1943–1955, 2016.

[26] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1251–1258, 2017.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

[27] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan, Q. V. Le, and H. Adam, "Searching for mobilenetv3," *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019.

[28] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Toward uniformed representation and acceleration for deep convolutional neural networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 11, pp. 2072–2085, 2016.

[29] G. Hegde, Siddhartha, and N. Kapre, "CaffePresso: Accelerating convolutional networkson embedded SoCs," *ACM Transactions on Embedded Computing Systems*, vol. 17, no. 1, pp. 1–26, 2017.

[30] X. Wei, Yu, Cody, Hao, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong, "Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs," *Proceedings of the 54th Annual Design Automation Conference 2017*, pp. 1–6, 2017.

[31] D. T. Nguyen, T. N. Nguyen, H. Kim, and H.-J. Lee, "A high-throughput and power-efficient FPGA implementation of YOLO-CNN for object detection," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 8, pp. 1861–1873, 2019.

[32] Y. Chen, T. Chen, Z. Xu, N. Sun, and O. Temam, "DianNao family: Energy-efficient hardware accelerators for machine learning," *Communications of the ACM*, vol. 59, no. 11, pp. 105–112, 2016.

[33] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.

[34] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 1–12, 2017.

[35] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-X: An accelerator for sparse neural networks," *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, 2016.

[36] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient inference engine on compressed deep neural network," *ACM SIGARCH Computer Architecture News*, no. 44, pp. 243–254, 2016.

[37] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1," *arXiv preprint arXiv:1602.02830*, 2016.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

[38] S. Liang, S. Yin, L. Liu, W. Luk, and S. Wei, "FP-BNN: Binarized neural network on FPGA," *Neurocomputing*, vol. 275, pp. 1072–1086, 2018.

[39] Z. He and D. Fan, "Simultaneously optimizing weight and quantizer of ternary neural network using truncated gaussian approximation," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 11 438–11 446, 2019.

[40] C. Zhu, S. Han, H. Mao, and W. J. Dally, "Trained ternary quantization," *5th International Conference on Learning Representations ICLR*, 2017.

[41] E. H. Lee, D. Miyashita, E. Chai, B. Murmann, and S. S. Wong, "Lognet: Energy-efficient neural networks using logarithmic computation," *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 5900–5904, 2017.

[42] S. Mittal, "A survey of FPGA-based accelerators for convolutional neural networks," *Neural Computing and Applications*, vol. 32, no. 4, pp. 1109–1139, 2018.

[43] E. Nurvitadhi, S. Subhaschandra, G. Boudoukh, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Ong Gee Hock, Y. T. Liew, K. Srivatsan, and D. Moss, "Can FPGAs beat GPUs in accelerating next-generation deep neural networks?" *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 5–14, 2017.

[44] R. Tapiador, A. Rios-Navarro, A. Linares-Barranco, M. Kim, D. Kadetotad, and J.-s. Seo, "Comprehensive evaluation of OpenCL-based convolutional neural network accelerators in Xilinx and Altera FPGAs," *Advances in Computational Intelligence. IWANN 2017. Lecture Notes in Computer Science*, no. 10306, 2017.

[45] S. Winograd, *Arithmetic complexity of computations*, 3rd ed., ser. CBMS-NSF regional conference series in applied mathematics. Philadelphia, Pa.: Society for Industrial and Applied Mathematics, 1990, vol. 33.

[46] Andrew Lavin and Scott Gray, "Fast algorithms for convolutional neural networks," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4013–4021, 2016.

[47] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, "Optimizing the convolution operation to accelerate deep neural networks on FPGA," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 7, pp. 1354–1367, 2018.

[48] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 580–587, 2014.

[49] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," *The IEEE International Conference on Computer Vision (ICCV)*, pp. 1440–1448, 2015.

[50] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection," *Proceedings of the IEEE international conference on computer vision*, pp. 2980–2988, 2017.

[51] P. Henderson and V. Ferrari, "End-to-end training of object class detectors for mean average precision," in *Asian Conference on Computer Vision*. Springer, 2016, pp. 198–213.

[52] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," *European conference on computer vision*, pp. 21–37, 2016.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

[53] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4510–4520, 2018.

[54] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 779–788, 2016.

[55] C.-Y. Wang, A. Bochkovskiy, and H.-Y. M. Liao, "Scaled-yolov4: Scaling cross stage partial network," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 13 029–13 038.

[56] P. Zhang, Y. Zhong, and X. Li, "Slimyolov3: Narrower, faster and better for real-time uav applications," *Proceedings of the IEEE International Conference on Computer Vision Workshops*, 2019.

[57] M. Tan, R. Pang, and Q. V. Le, "Efficientdet: Scalable and efficient object detection," *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 10 781–10 790, 2020.

[58] NVIDIA, "Nvidia-turing-architecture-whitepaper," 2018. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

[59] ——, "Nvidia-ampere-ga102-gpu-architecture-whitepaper-v1," 2020. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/geforce/ampere/pdf/NVIDIA-ampere-GA102-GPU-Architecture-Whitepaper-V1.pdf

[60] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "DaDianNao: A machine-learning supercomputer," *47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 609–622, 2014.

[61] NVIDIA, "Nvidia tesla V100 GPU architecture," 2017. [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[62] S. I. Venieris, A. Kouris, and C.-S. Bouganis, "Toolflows for mapping convolutional neural networks on FPGAs: A survey and future directions," *ACM Computing Surveys (CSUR)*, no. 51, pp. 1–39, 2018.

[63] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1–9, 2015.

[64] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *International Conference on Learning Representations*, 2015.

[65] NVIDIA, "Nvidia tegra 4 family GPU architecture," 2015. [Online]. Available: https://www.nvidia.com/docs/IO/116757/Tegra_4_GPU_Whitepaper_FINALv2.pdf

[66] ——, "Jetson AGX Xavier and the new era of autonomous machines." [Online]. Available: https://info.nvidia.com/rs/156-OFN-742/images/Jetson_AGX_Xavier_New_Era_Autonomous_Machines.pdf

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

[67] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning: Acm sigarch computer architecture news," vol. 42, no. 1, pp. 269–284, 2014.

[68] T. Wang, C. Wang, X. Zhou, and H. Chen, "A survey of FPGA based deep learning accelerators: Challenges and opportunities," *arXiv preprint arXiv:1901.04988*, pp. 1–10, 2018.

[69] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang, "[dl] a survey of FPGA-based neural network inference accelerators," *ACM Transactions on Reconfigurable Technology ans Systems*, vol. 12, no. 1, 2019.

[70] S. I. Venieris and C.-S. Bouganis, "fpgaConvNet: Mapping regular and irregular convolutional neural networks on FPGAs," *IEEE transactions on neural networks and learning systems*, vol. 30, no. 2, pp. 326–342, 2019.

[71] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 161–170, 2015.

[72] U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling, and G. R. Chiu, "An opencl™ deep learning accelerator on Arria 10," *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 55–64, 2017.

[73] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi *et al.*, "A configurable cloud-scale DNN processor for real-time ai," *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1–14, 2018.

[74] J. Wang and S. Gu, "Fpga implementation of object detection accelerator based on vitis-ai," in *2021 11th International Conference on Information Science and Technology (ICIST)*. IEEE, 2021, pp. 571–577.

[75] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, "DNNBuilder: an automated tool for building high-performance DNN hardware accelerators for FPGAs," *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, 2018.

[76] S. H. Mohamed, T. E. H. El-Gorashi, and J. M. H. Elmirghani, "A survey of big data machine learning applications optimization in cloud data centers and networks," *arXiv preprint arXiv:1910.00731*, 2019.

[77] H. Halawa, H. A. Abdelhafez, A. Boktor, and M. Ripeanu, "Nvidia jetson platform characterization," in *European Conference on Parallel Processing*. Springer, 2017, pp. 92–105.

[78] B. Liu, X. Chen, Y. Wang, Y. Han, J. Li, H. Xu, and X. Li, "Addressing the issue of processing element under-utilization in general-purpose systolic deep learning accelerators," in *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, 2019, pp. 733–738.

[79] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to fpgas," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Erklärung zur Verfassung der Arbeit

Hiermit erkläre ich, dass die vorliegende Arbeit gemäß dem Code of Conduct – Regeln zur Sicherung guter wissenschaftlicher Praxis (in der aktuellen Fassung des jeweiligen Mitteilungsblattes der TU Wien), insbesondere ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel, angefertigt wurde. Die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In– noch im Ausland in gleicher oder in ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Vienna, Austria 27.10.2021

Lukas Baischer

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Appendix A

.1 Dataflow visualizations

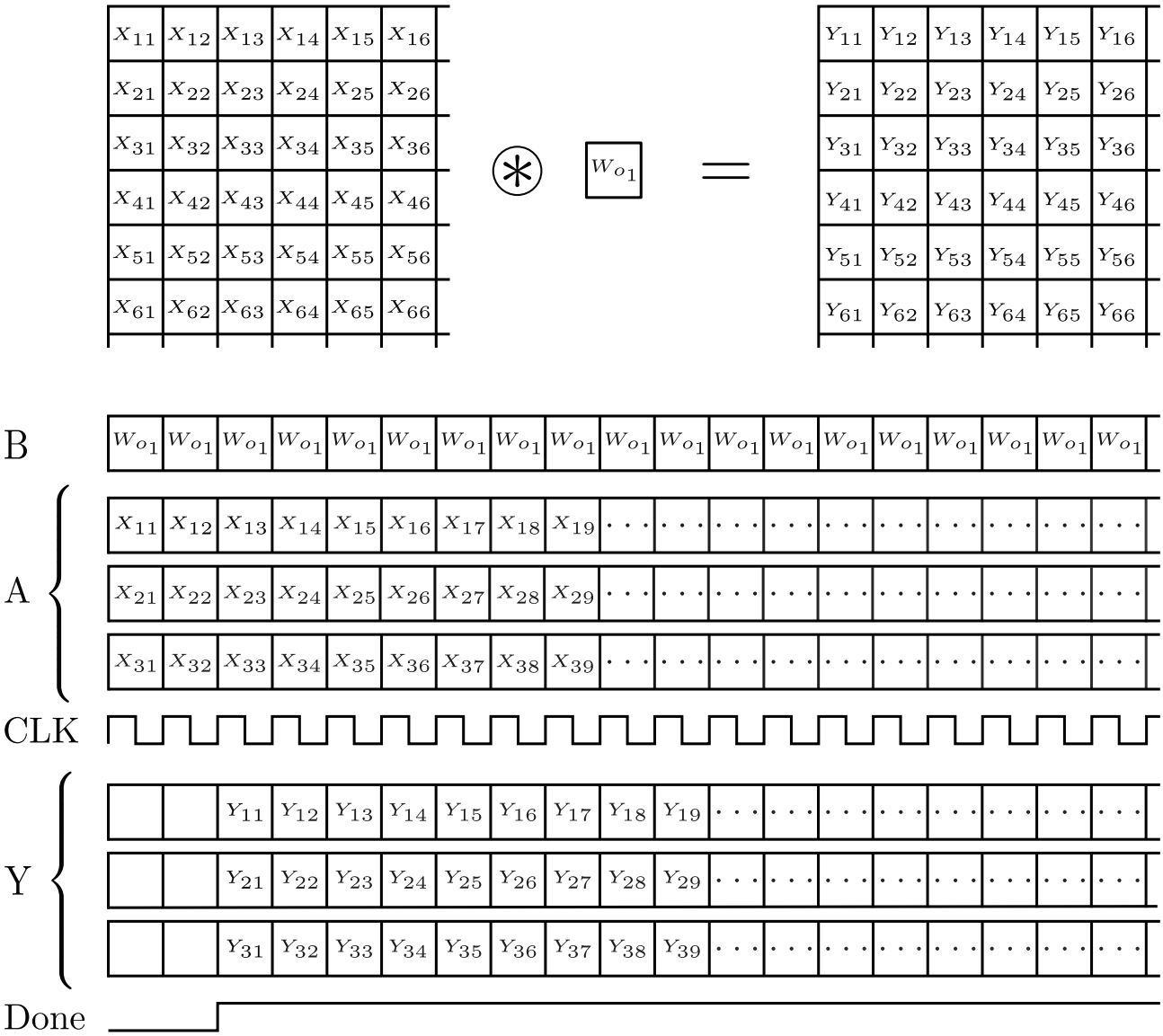


Figure 1: Dataflow for 1×1 convolutions. The colours highlight, which input data is used to compute a specific output pixel. Due to a the delay of 3 clock cycles caused by the pipelining of the DSP, 3 output channels are computed alternating.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

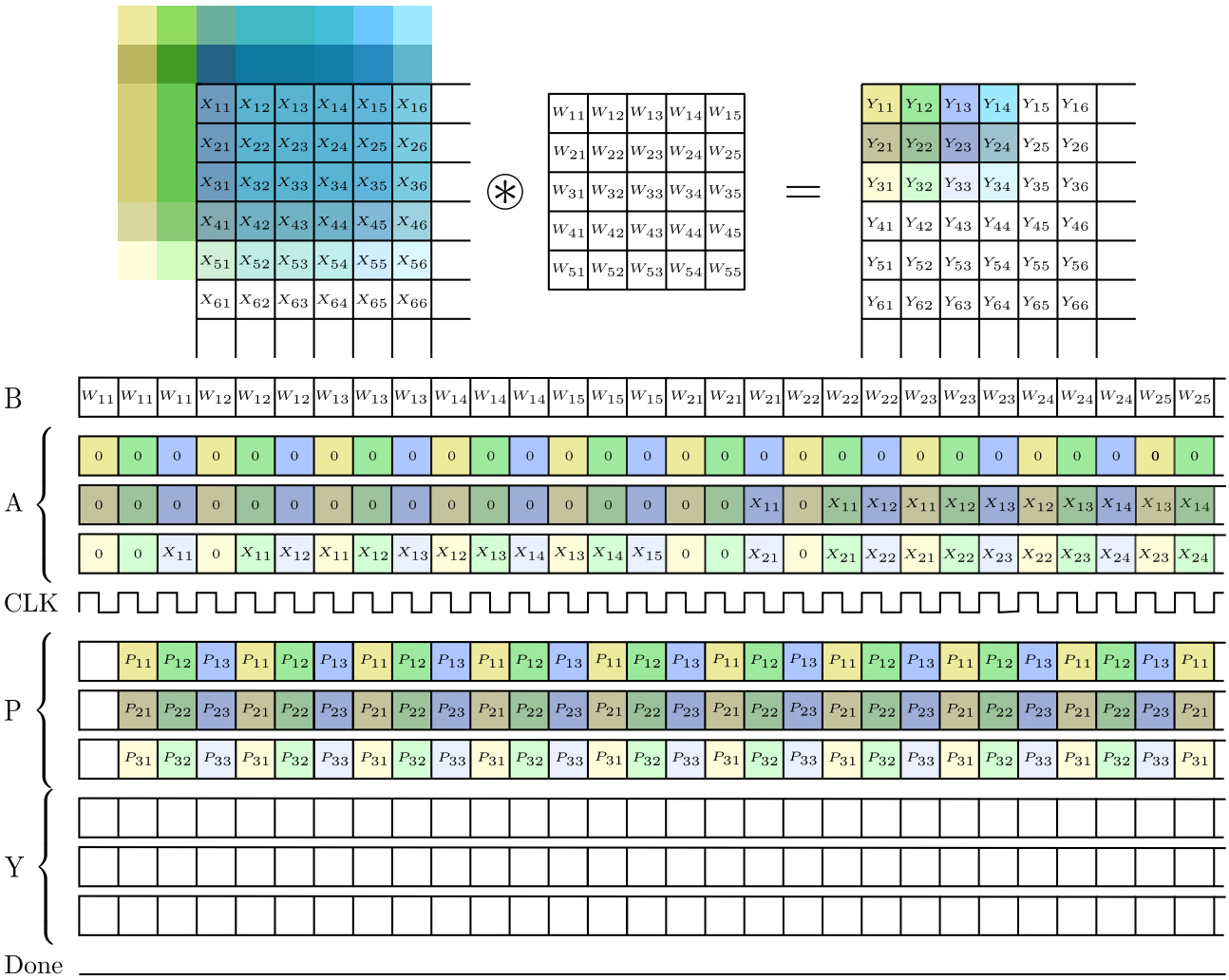


Figure 2: Dataflow for 5×5 convolutions. The colours highlight, which input data is used to compute a specific output pixel. Due to a the delay of 3 clock cycles caused by the pipelining of the DSP, 3 output pixels are computed alternating.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

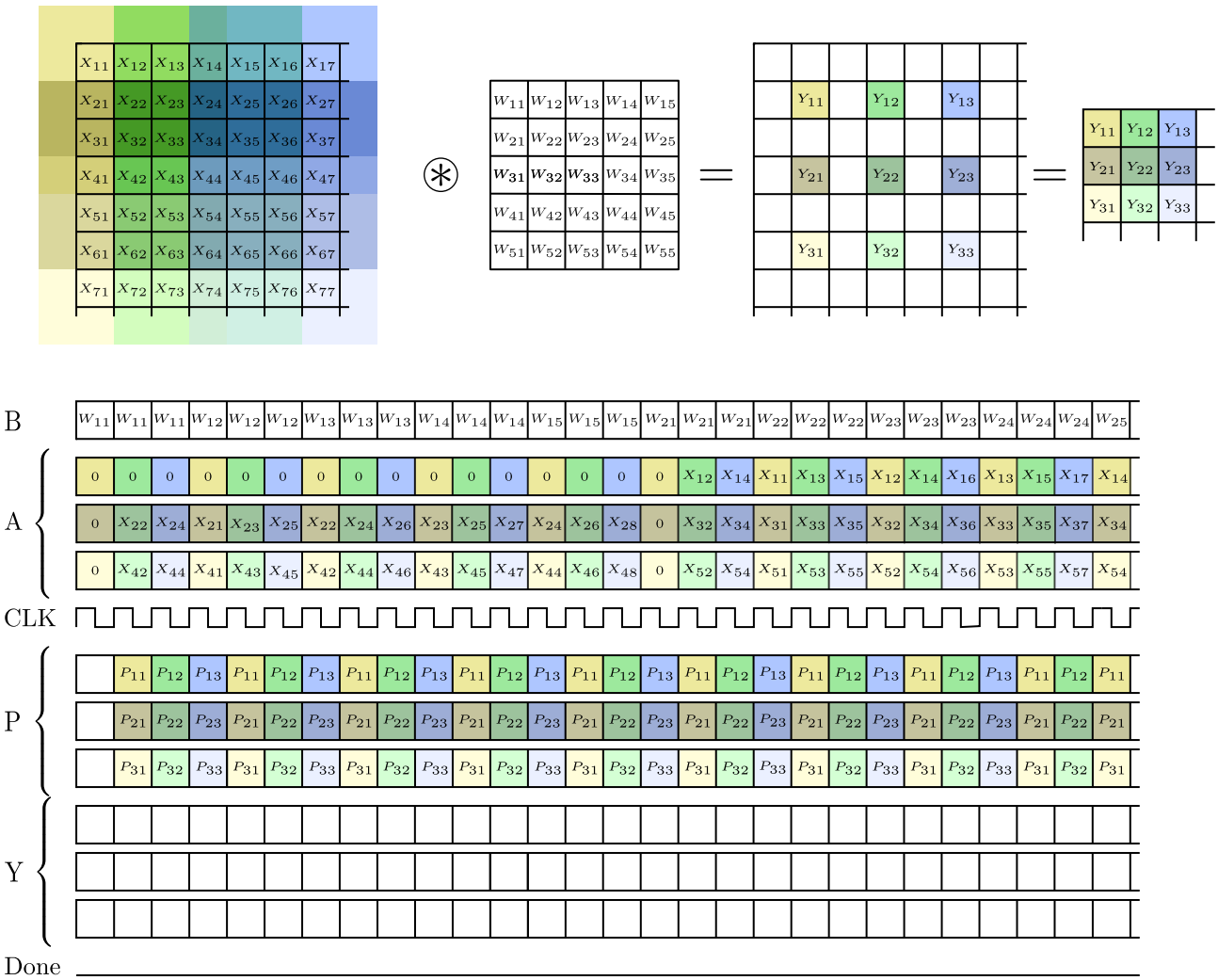


Figure 3: Dataflow for 5×5 convolutions width stride 2. The colours highlight, which input data is used to compute a specific output pixel. Due to a the delay of 3 clock cycles caused by the pipelining of the DSP, 3 output pixels are computed alternating.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

.2 YOLOv3-tiny application

```
import numpy as np
import cv2
import pathlib
import argparse
import shutil
import timeit
from elevate import elevate
import os

from intuitus_nn import Framebuffer, Camera
import intuitus_nn as nn
from core.utils import YoloLayer, filter_boxes, draw_bbox_nms, nms, read_class_names

def _create_parser():
    parser = argparse.ArgumentParser()
    parser.add_argument('--command_path',  type=str, default='./intuitus_commands/yolov3-tiny-
                                                        commands', help='path to intuitus command
                                                        files')
    parser.add_argument('--classes_path',  type=str, default='./data/classes/coco.names', help='path
                                                         to class name file')
    parser.add_argument('--image',  type=str, default='./cam_data/000000000069.jpg', help='path to
                                                        input image')
    parser.add_argument('--output',  type=str, default='./fmap_out', help='path to output')
    parser.add_argument('--tiny', type=bool, default=True, help='is yolo-tiny or not')
    parser.add_argument('--size', type=int, default=416, help='define input size of export model')
    parser.add_argument('--conf_thres', type=float, default=0.1, help='define confidence threshold')
    parser.add_argument('--iou_thres', type=float, default=0.4, help='define iou threshold')
    parser.add_argument('--score', type=float, default=0.15, help='object confidence threshold')
    parser.add_argument('--use_cam', action='store_true', help='use camera for input img')
    parser.add_argument('--use_fb', action='store_true', help='write output to framebuffer')
    parser.add_argument('--print_net', action='store_true', help='print network')
    parser.add_argument('--print_layer_info', type=int, default=None, help='print layer dma info.
                                                        Expects layer number to be printed.')
    parser.add_argument('--save_result_boxes', action='store_true', help='save network output boxes
                                                        as .npy file')
    parser.add_argument('--not_execute', action='store_true', help='print network')
    parser.add_argument('--iterations', type=int, default=1, help='execution iterations')

    return parser.parse_args()

def _yolov3_tiny_config():
    yolo_lb = { 'anchors':np.array([[81,82],  [135,169],  [344,319]]),
                'stride': 32,
                'classes':80}
    yolo_mb = { 'anchors':np.array([[10,14],  [23,27],  [37,58]]),
                'stride': 16,
                'classes':80}
    yolo_conf = {   'lb':yolo_lb,
                    'mb':yolo_mb}
    return yolo_conf
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

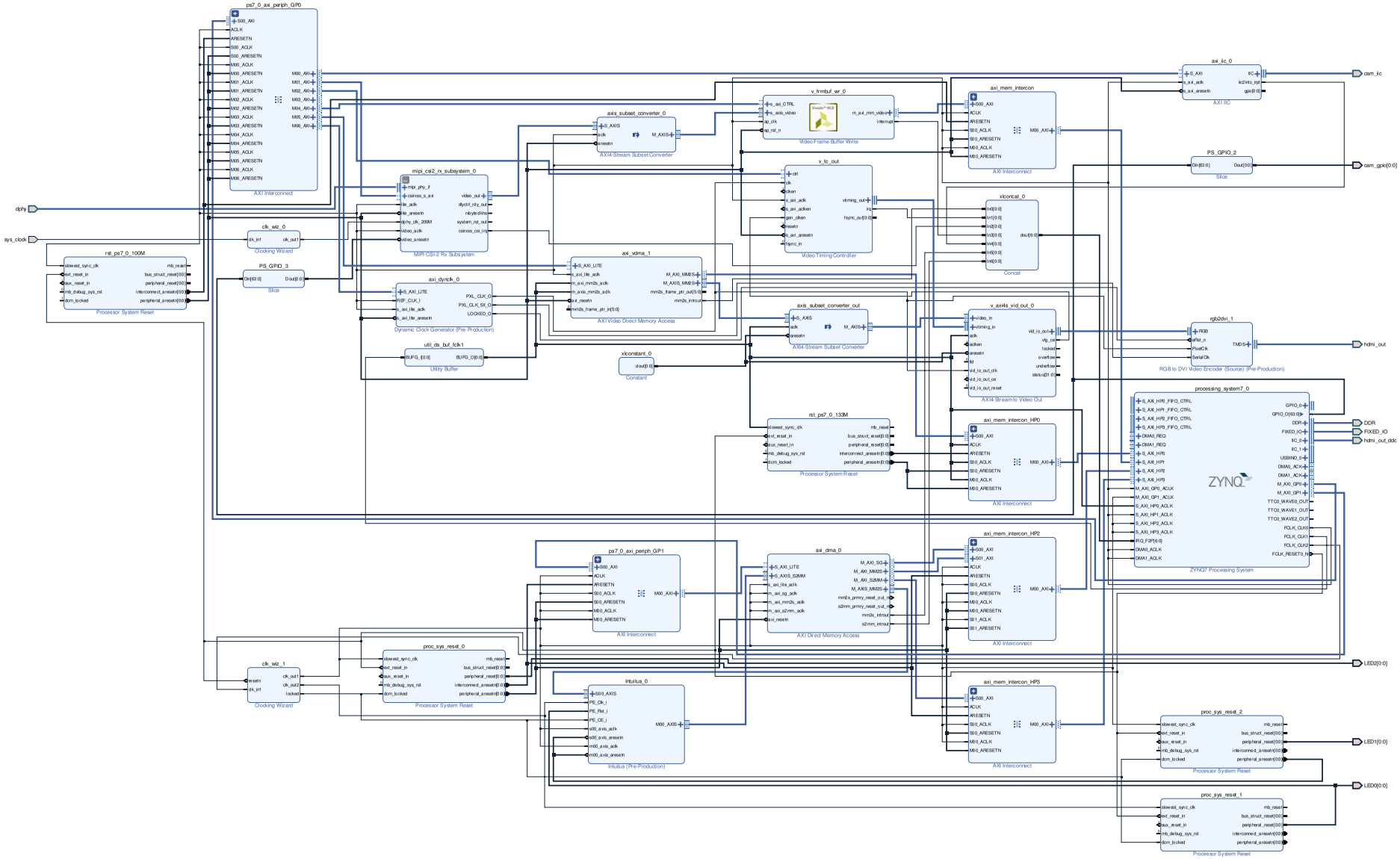
```
45
46 def main(flags):
47     if os.getuid() != 0:
48         try: elevate() # get root privileges. Required to open kernel driver
49         except: raise Exception("Root permission denied")
50     exectue_net = not flags.not_execute
51     input_size = flags.size
52     image_path = pathlib.Path(__file__).absolute().parent / flags.image
53     command_path = pathlib.Path(__file__).absolute().parent / flags.command_path
54     class_name_path = pathlib.Path(__file__).absolute().parent / flags.classes_path
55     out_path = pathlib.Path(__file__).absolute().parent / flags.output
56     if out_path.exists():
57         shutil.rmtree(out_path, ignore_errors=False, onerror=None)
58     out_path.mkdir(parents=True, exist_ok=True)
59
60     if flags.tiny:
61         yolo_config = _yolov3_tiny_config()
62     else:
63         raise NotImplementedError("Add config for non tiny implementaion")
64     result_scale = 2.0**-4.0
65
66     Net = nn.Sequential(command_path)
67     buffer = Net.input(3,input_size,input_size)
68
69     #%% backbone
70     buffer = Net.conv2d(buffer,16,(3,3),max_pooling=True,command_file='conv2d_0.npz') #208
71     buffer = Net.conv2d(buffer,32,(3,3),max_pooling=True,command_file='conv2d_1.npz') #104
72     buffer = Net.conv2d(buffer,64,(3,3),max_pooling=True,command_file='conv2d_2.npz') #52
73     buffer = Net.conv2d(buffer,128,(3,3),max_pooling=True,command_file='conv2d_3.npz') #26
74     buffer = Net.conv2d(buffer,256,(3,3),command_file='conv2d_4.npz') #26
75     route_1 = buffer
76     buffer = Net.maxpool2d(buffer, strides=(2,2)) # Software pooling (CPU)
77     buffer = Net.conv2d(buffer,512,(3,3),command_file='conv2d_5.npz') #13
78     buffer = Net.maxpool2d(buffer, strides=(1,1)) # Software pooling (CPU)
79     buffer = Net.conv2d(buffer,1024,(3,3),command_file='conv2d_6.npz') #13
80
81     #%% head
82     buffer = Net.conv2d(buffer,256,(1,1),command_file='conv2d_7.npz') #13
83     lobj = buffer
84     conv_lobj_branch = Net.conv2d(lobj,512,(3,3),command_file='conv2d_8.npz') #13
85     conv_lbbox = Net.conv2d(conv_lobj_branch,256,(1,1),command_file='conv2d_9.npz') #13
86     buffer = Net.conv2d(buffer,128,(1,1),command_file='conv2d_10.npz') #13
87     buffer = Net.upsample(buffer)
88     buffer = Net.concat(buffer,route_1) #26
89     mobj = buffer
90     conv_mobj_branch = Net.conv2d(mobj,256,(3,3),command_file='conv2d_11.npz') #26
91     conv_mbbox = Net.conv2d(conv_mobj_branch,256,(1,1),command_file='conv2d_12.npz') #26
92
93     Net.output(conv_lbbox)
94     Net.output(conv_mbbox)
95
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

```
96      Yolo_lb = YoloLayer(yolo_config['lb']['anchors'],
97                          yolo_config['lb']['classes'],
98                          yolo_config['lb']['stride'])
99      Yolo_mb = YoloLayer(yolo_config['mb']['anchors'],
100                         yolo_config['mb']['classes'],
101                         yolo_config['mb']['stride'])
102     if flags.print_net:
103         Net.summary()
104     if flags.print_layer_info != None:
105         Net.print_layer_dma_info(flags.print_layer_info)
106
107     if flags.use_cam:
108         cam = Camera('/dev/video0')
109     if flags.use_fb:
110         fb = Framebuffer('/dev/fb0')
111
112     for i in range(flags.iterations):
113         if flags.use_cam:
114             img_start = timeit.default_timer()
115             status, original_image = cam.capture()
116             img_cap = timeit.default_timer()
117             original_image = cv2.cvtColor(original_image,cv2.COLOR_YUV2BGR_Y422)
118             original_image = cv2.flip(original_image, 0)
119             original_image[...,1] = np.uint8(original_image[...,1]*0.66) # fix green channel error
120             image_data = cv2.resize(original_image, (input_size, input_size))
121             b,g,r = cv2.split(image_data)
122             image_data = np.stack([r,g,b]).astype(np.uint8)
123             img_time = timeit.default_timer()
124             print(  "Img capture time: {} ms.\n".format((img_cap-img_start)*1000) + \
125                     "Img resize time: {} ms".format((img_time-img_cap)*1000))
126
127         elif '.npz' in flags.image:
128             img_npz = np.load(str(image_path),allow_pickle=True)
129             image_data = img_npz['img'].astype(np.uint8)
130         elif '.npy' in flags.image:
131             image_data = np.load(str(image_path),allow_pickle=True)
132         else:
133             original_image = cv2.imread(str(image_path))
134             image_data = cv2.resize(original_image, (input_size, input_size))
135             b,g,r = cv2.split(image_data)
136             image_data = np.stack([r,g,b]).astype(np.uint8)
137
138         if exectue_net:
139             nn_start = timeit.default_timer()
140             fmap_out = Net(image_data)
141             cnn_time = timeit.default_timer()
142
143             pred_lb = Yolo_lb(fmap_out[0][:255,...]*result_scale)
144             pred_mb = Yolo_mb(fmap_out[1][:255,...]*result_scale)
145             inf_out = np.concatenate((pred_lb,pred_mb),axis=0)
146             yolo_layer_time = timeit.default_timer()
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

```
147
148             boxes, pred_conf, classes = filter_boxes(inf_out,flags.conf_thres)
149             if not boxes.shape[0]:
150                 continue
151             best_bboxes = nms(boxes, pred_conf, classes, iou_threshold = flags.iou_thres,
152                             score=flags.score,method='merge')
153             if not best_bboxes.shape[0]:
154                 continue
155             classes = read_class_names(str(class_name_path))
156             image_data = np.moveaxis(image_data,0,-1).astype(np.uint8).copy('C')
157             image = draw_bbox_nms(image_data, best_bboxes,classes)
158
159             nms_time = timeit.default_timer()
160
161             print(  "CNN Excution time: {} ms. \n".format((cnn_time-nn_start)*1000) + \
162                     "YOLO layer(numpy): {} ms. \n".format((yolo_layer_time-cnn_time)*1000) + \
163                     "NMS (numpy): {} ms. \n".format((nms_time-yolo_layer_time)*1000) + \
164                     "Full time: {} ms".format((nms_time-nn_start)*1000))
165             if flags.save_result_boxes:
166                 outfile_name = 'best_bboxes.npy'
167                 np.save(str(out_path / outfile_name),best_bboxes)
168
169         if flags.use_fb:
170             plot_start= timeit.default_timer()
171             img_bgr = cv2.cvtColor(image,cv2.COLOR_RGB2BGR)
172             screen_size = fb.get_screensize()
173             img_bgr = cv2.resize(img_bgr,(screen_size[1],screen_size[0]))
174             fb.show(img_bgr,0)
175             plot_time = timeit.default_timer()
176             print(  "Plot time: {} ms. \n".format((plot_time-plot_start)*1000))
177
178         else:
179             image = cv2.cvtColor(np.array(image), cv2.COLOR_BGR2RGB)
180             outfile_name = 'detect.png'
181             cv2.imwrite(str(out_path / outfile_name), image)
182
183 if __name__ == '__main__':
184     flags = _create_parser()
185     main(flags)
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.